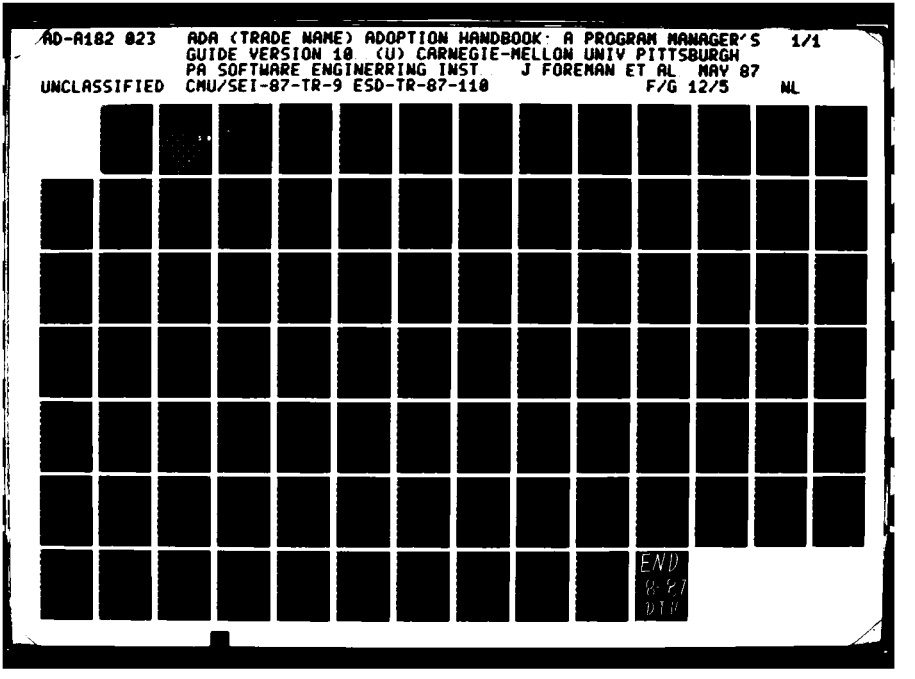
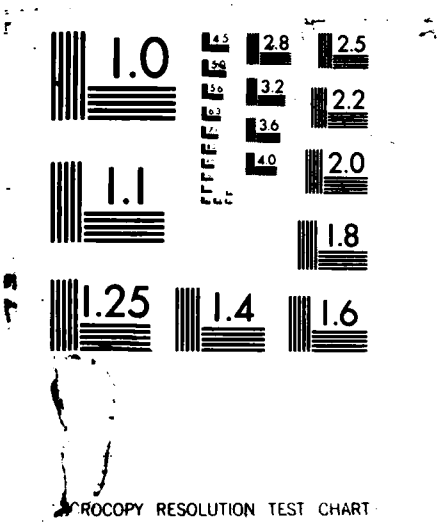

AD-A182 023 ADA (TRADE NAME) ADOPTION HANDBOOK: A PROGRAM MANAGER'S 1/1
GUIDE VERSION 1.0 (U) CARNEGIE-MELLON UNIV PITTSBURGH
PA SOFTWARE ENGINEERRING INST J FOREMAN ET AL MAY 87
UNCLASSIFIED CMU/SEI-87-TR-9 ESD-TR-87-110 F/G 12/5 NL
END
8-87
DTIC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
ROCOPY RESOLUTION TEST CHART

Technical Report

CMU/SEI-87-TR-9
ESD-TR-87-110

Carnegie-Mellon University
Software Engineering Institute

Ada® Adoption Handbook: A Program Manager's Guide

John Foreman
John Goodenough

May 1987

87

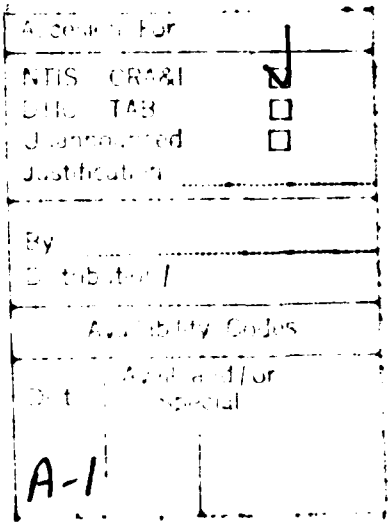
Technical Report
CMU/SEI-87-TR-9
ESD-TR-87-110
May 1987

Ada® Adoption Handbook: A Program Manager's Guide

Version 1.0

John Foreman
John Goodenough

Ada Handbook Project



Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler
SEI Joint Program Office

This work was sponsored by the U.S. Department of Defense.

Table of Contents

List of Figures

List of Tables

Acknowledgements

As part of the development of this document, we received reviews and comments from a wide spectrum of people:

- Ada technologists
- Department of Defense (DoD) policy makers
- mission-critical computer resource (MCCR) system developers and support organizations
- Department of Defense and industry program managers

Many organizations provided comments on this document. We thank them all. We are also grateful to the following individuals who gave so generously of their time:

Bryce Bardin, Hughes Aircraft Company
Grady Booch, Rational
Lt. Col. Mike Borky, U.S. Air Force
Craig Brooks, U.S. Army
Capt. Jim Cardow, U.S. Air Force
Robert M. Earnest, Jr., U.S. Air Force
Don Evans, Tartan Laboratories
Col. Jack Ferguson, U.S. Air Force
Maj. Steve Frith, U.S. Army
Mark Gerhardt, Oasys Federal Systems
Carolyn Gannon, Consultant
Lt. Col. Dave Hammond, U.S. Air Force
Charles (Bud) Hammons, Texas Instruments
Rich Hilliard, MITRE
1st Lt. Kurt W. Hoyt, U.S. Air Force
Tony Jordano, IBM
Maj. Mike Kaye, U.S. Air Force
Lt. Col. John Leary, U.S. Air Force
Steven Litvintchouk, MITRE
Maj. Robert Lyons, U.S. Air Force
Frank McGarry, NASA Goddard Space Flight Center
Freeman Moore, Texas Instruments
Lt. Cmdr. Philip Myers, Ada Joint Program Office
Leland Page, Federal Aviation Administration
Capt. Victoria Rhoden, U.S. Air Force
Dudrey Smith, Lear Siegler, Inc.
Raymond Szymanski, U.S. Air Force
Valentin Tirman, Ford Aerospace & Communications
Col. Paul Wickliffe, U.S. Army, Retired
William Wilder, U.S. Navy

The authors express special appreciation to the following individuals for their exacting reviews and pragmatic insights:

- Maj. Daniel Burton, U.S. Air Force, Software Engineering Institute Joint Program Office
- Patricia Oberndorf, U.S. Navy, Naval Ocean Systems Center, San Diego

Additionally, Sandra Bond of the Software Engineering Institute Information Services group has been invaluable for trying to "make every word count."

Abstract

The Ada Adoption Handbook provides program managers with information about how best to tap Ada's strengths and manage this new software technology. Although the issues are complex, they are not all unique to Ada. Indeed, many of these issues must be addressed when using any language for building sophisticated systems. The handbook addresses the advantages and risks inherent in adopting Ada. Significant emphasis has been placed on providing information and suggesting methods that will help program and project managers succeed in adopting Ada across a broad range of application domains.

The handbook focuses on the following topics: program management issues including costs and technical and program control; Ada's goals and benefits; software tools with emphasis on compiler validation and quality issues; the state of Ada technology as it relates to system engineering; the application of special purpose languages; issues related to mixing Ada with other languages; possible productivity benefits resulting from software reuse; and implications for education and training.

1. Introduction

1.1. Purpose and Scope

> *Ada is the official Department of Defense programming language for developing mission-critical and embedded applications. This handbook presents program managers with information to make effective use of Ada.*

The Department of Defense (DoD) has declared Ada the required language for developing mission-critical and embedded applications. This handbook provides program managers with information about how best to tap Ada's strengths and manage this new software technology.[1] Although the issues are sometimes complex, the reader should not infer that adopting Ada is therefore complex. Similar issues must be addressed when using any language for building sophisticated systems.

This handbook addresses the advantages and risks inherent in adopting Ada. Significant emphasis has been placed on providing information and suggesting methods that will help program and project managers succeed in adopting Ada. This handbook should not be construed as contrary to DoD policy.

This handbook has been written for use across many application domains and covers a large number of fundamental Ada issues. However, it is not possible for this handbook to address issues that are unique to individual application domains (for example, ground-based command and control systems or embedded avionics). Special adaptation or application is left to application domain specialists.

[1]Detailed technical documents for software managers and lead technical personnel will be available beginning in 1988.

1.2. Organizational Overview

This section describes the organization of this handbook.

This handbook is organized as stand-alone chapters that address program managers' questions about Ada and provide plans for adopting and inserting Ada into an organization. The chapters are:

1. **Introduction**: This chapter. Includes the purpose of this handbook and provides pointers for using the handbook.
2. **Program Manager Considerations**: Commonly asked questions, succinct answers, and, where needed, pointers to more detailed information. Topics include general information, costs, and technical and program control issues. This chapter can be used (among other purposes) as a executive summary for the rest of the handbook and to review particular points.
3. **The Need for Ada**: Ada's goals and benefits.
4. **Software Production Technology**: An overall view of software tools, with particular emphasis on compiler validation and compiler quality issues.
5. **Ada Maturity and Applicability**: An overall view of the current state of Ada technology and discussion of system engineering.
6. **Using Special-Purpose Languages**: The application of special-purpose languages such as ATLAS, simulation programming languages, LISP (or other artificial intelligence languages), and fourth-generation languages.
7. **Mixing Ada with Other Languages**: Possible approaches for dealing with existing operational systems, including language translation and hybrid (mixed language) systems.
8. **Software Reuse and Ada**: A discussion of the *history*, *current efforts*, current issues, and benefits expected from software reuse.
9. **Learning Ada**: Training implications and education issues.

The following appendices are also included:

I. Ada Working Groups and Agencies

II. Programs Using Ada

III. Ada Textbooks

IV. Ada Compilers for Target Processors

1.3. Tips for Readers

This section highlights techniques for quick and efficient use of this handbook.

In addition to the Question and Answer approach used in Chapter 2, several other techniques have been used to help the reader make maximum use of this handbook:

- **Software Development Tutorial**: The terms and concepts of software development discussed in this handbook are defined in Section 4.1. Readers unfamiliar with these terms and concepts are strongly encouraged to read that section first.
- **Summaries**: A summary begins each major section. Each summary is centered and italicized for easy identification.
- **Action Plans**: Actions to facilitate the adoption of Ada are presented at the end of some major sections.
- **Bold and Bullets**: Major points are emphasized by using bold headings within bulleted lists. The major points are then followed by detailed discussion.

Since Ada technology is rapidly advancing, some information in this handbook is time sensitive. Examples are Section 5.2, Scenarios for Ada Use, and Appendix IV, the selected list of validated compilers. This handbook will be re-issued periodically to present up-to-date information about Ada. Program managers should ensure that they have access to the most up-to-date information as part of their decision making process.

The following topics are not covered (or are given only limited coverage) in this handbook:

- details of the Ada language
- details of particular methodologies such as object-oriented design (OOD), process abstraction (PAMELA), and structured analysis
- use of Ada as a program design language (PDL).

For those interested in learning about these topics, see the appendices.

2. Program Manager Considerations

Ada offers considerable advantages over other languages used for software development. Still, because the technology supporting Ada is still evolving, program managers may have questions about this new, maturing technology. Questions fall into several categories: general, costs, technical issues, program control, and getting help.

This handbook provides the information program managers need to make well-informed decisions about using Ada. Some typical questions and answers are presented on the following pages. Where needed, pointers to supplemental information and appropriate actions are provided.

2.1. General

The following questions are covered in this section:

- What is Ada and why was it developed?
- Who manages Ada?
- What are the advantages of using Ada?

Question: What is Ada and why was it developed?

Answer: Software development and post-deployment support are increasingly recognized as serious problems for mission-critical system development. Ada is the DoD programming language (defined in ANSI/MIL-STD-1815A) designed to address these problems by:

- reducing the number of DoD programming languages so effort can be focused on making one language work well, and
- encouraging the use of modern software development methods, thereby reducing the costs and risks of software development and facilitating improved maintenance during post-deployment support.

Ada is an American, NATO, and international standard, as well as a DoD standard.

See: 3.1.

Question: Who manages Ada?

Answer: The Ada language effort is managed by the Ada Joint Program Office (AJPO), which is responsible for, among other activities:

- maintaining Ada as military, ANSI, and FIPS standards,
- developing compiler validation procedures and guidelines,
- certifying Ada validation facilities that offer validation of Ada compilers, and
- coordinating DoD training and education activities.

The AJPO reports to the Deputy Undersecretary of Defense for Research and Advanced Technology (DUSDR&AT).

See: Appendix I.

Question: What are the advantages of using Ada?

Answer: Ada's overall advantage can be summarized as increased quality per dollar spent. Specifically, benefits are expected and have been reported in productivity, software portability, people portability, and software maintainability and reliability [10, 6, 14, 31]. Additionally, Ada will provide a common basis for tools and methodologies along with new capabilities for managing system complexity.

See: 3.1, 3.2.

2.2. Costs

The following questions are covered in this section:

- Will Ada save the DoD money?
- If a program requires a new or modified compiler, what are the cost, schedule, and funding implications?
- What impact will using Ada have on hardware requirements?

Question: Will Ada save the DoD money?

Answer: Cost reductions will accrue to the DoD through the aggregate use of a single, modern, high-order language. Because Ada incorporates new technology, individual programs will incur some one-time start-up costs for software tools, training, and development hardware (although in some cases, these costs may be shared by several programs). However, in the long run, the use of Ada is expected to reduce individual program development and maintenance costs because the language encourages the use of better software development methods.

Question: If a program requires a new or modified compiler, what are the cost, schedule, and funding implications?

Answer: Based on the official AJPO list (1 May 1987), there were 78 validated compilers generating code for at least 30 different machines. Since Ada compilers are available for a wide variety of computers, there is some degree of choice possible in the selection of target processors and compilers. As such, it should not be necessary to build a completely new compiler. Two other actions may be necessary:

- **Retargetting**: Extending a compiler to generate code for a new target computer. This process is also known as building a new code generator or compiler back end. (A compiler consists of at least two parts — the front end and the back end. The front end, which is also called the machine independent portion (MI), contains the components that are independent of the characteristics of the target computer and can be used in common for many target computers. The front end generally will not have to be modified during the retargetting process. The back end contains the components that depend on the characteristics of the target computer and which therefore must be designed specifically for each target computer.)
- **Program-Specific Enhancements**: Modifying a compiler to meet program-specific performance or resource constraints. Among such activities would be improving the generated object code, tailoring the run-time software, and adding additional debugging capabilities.

These activities are likely to require the use of program funds; there is generally no DoD-wide funding for these efforts. While there are some exceptions (for example, the Navy is funding the development of standard compilers for their AN/UYK-43, AN/UYK-44, and AN/AYK-14 machines), the current trend is to use commercially developed products because of their diversity and quality.

The costs of retargetting and program-specific enhancements are likely to be small in relation to the total hardware and software costs for a program. Guidance regarding cost and schedule follows:

- **Retargetting**: Some cost and schedule ranges are $750K to $1750K and 9 to 15 months' development time. Additional time should be allowed to mature the new code generator. These ranges may be affected by:
  - the compiler technology used,
  - the difficulty of the code generation task (see Section 2.3 for a brief discussion of hardware considerations),
  - economic factors such as the potential marketplace for the completed compiler and the competitive posture of the software houses bidding for the work.
- **Program-Specific Enhancements**: Cost and schedule depend on the requirements and technical objectives of the effort.

See: 4.1, 4.2, 4.4, 4.5, 5.3, 5.4, Appendix IV.

Question: What impact will using Ada have on hardware requirements?

Answer: This question requires an examination of both host (or development) computer resources and target computer resources.

- **Host computer**: Ada compilers require more host computing resources than do compilers for most other languages. However, Ada compilers detect many more programming errors (thereby saving time during software integration) and provide greater levels of support for building large software systems than do other languages.
- **Target computer**: Ada should be as efficient as other languages (and perhaps more so) as compilers mature and implementors support Ada language features that aid in generating efficient code. Currently, however, compilers targeted to resource-constrained machines have not reached this level of maturity. Hardware selection should be made such that processor speed or memory size can be increased if necessary.

See: 4.2.2, 4.2.2.1, 4.2.2.2, 4.5, 5.3, 5.4.

2.3. Technical Issues

The following questions are covered in this section:

- What is the current status of Ada compilers? What criteria should be used to determine which compilers are most appropriate for a program?
- What is validation? How does the requirement to use only validated compilers affect a program that must modify its compiler? Can this requirement cause schedule slippage?
- What is the purpose of an Ada Program Design Language (PDL)? What are the advantages and disadvantages?
- Can Ada be used on MIL-STD-1750A computers? On Intel 8086? On the Motorola 68000 family? Are any of these processors more appropriate to use with Ada?

- Should every processor in a weapon system be programmed in Ada?
- Can Ada be used in distributed applications?
- What is run-time software and why are run-time issues so important to Ada?
- What is an APSE? Is an APSE required to use Ada?
- What is the significance of the Common APSE Interface Set (CAIS) development?
- What is software portability? Why is this an important Ada advantage?
- Can Ada be mixed with other languages?
- Are there risks in automatically translating existing systems to Ada?
- Can database management packages and fourth-generation languages still be used?
- Why is there interest in software reuse? Can Ada help?

Question: What is the current status of Ada compilers? What criteria should be used to determine which compilers are most appropriate for a program?

Answer: Ada compilers are suitable now for applications that run on general-purpose computers with no severe memory or time-critical performance constraints. For performance and resource-constrained applications (avionics, fire control, etc.), Ada compilers and run-time systems are just beginning to appear. Each program will have to evaluate the quality and maturity of compilers based on compile-time efficiency, object-code efficiency, additional compiler services, and support for embedded system requirements. The Ada Compiler Evaluation Capability (ACEC), currently under development by the AJPO through the Avionics Laboratory at Wright Patterson Air Force Base, will provide a common method for evaluating Ada compilers. The first release of the ACEC is scheduled for July 1988.

See: 4.2.2, 5, Appendix I.

Question: What is validation? How does the requirement to use only validated compilers affect a program that must modify its compiler? Can this requirement cause schedule slippage?

Answer: Validation is a process developed by the AJPO to test an Ada compiler's compliance with the language definition (ANSI/MIL-STD-1815A). To achieve validation, an Ada compiler must pass a standard test suite known as the Ada Compiler Validation Capability (ACVC).[2] The validation process does *not* address performance areas such as compiler speed or efficiency of the generated code, so validation does *not* imply that a compiler is suitable (in terms of quality or features) for use by a specific program. According to the current validation guidelines [4], changes can be made to a compiler as long as they do not change the language. The validation tests should be run against a customized compiler to ensure there are no deviations from the standard. The program manager decides when to revalidate a program's compiler. For further details about validation policy, see [4].

See: 4.2.1.

[2]The compiler validation process is somewhat analogous to the SEAFAC testing, which ensures compliance of MIL-STD-1750A computers to their standard.

Question: What is the purpose of an Ada Program Design Language (PDL)? What are the advantages and disadvantages?

Answer: A PDL is a formal notation used to capture design decisions. Ada's features for structuring and organizing programs are also usable in design to describe software architecture (the structure of a system), its interfaces, and its overall behavior, as well as to document that design. Any design language should be used as a part of an overall software methodology. When placing an Ada PDL on contract, attention should be paid to how well it is integrated into the contractor's methodology. Many ongoing procurements have called for the use of an Ada PDL, and DoD directive 3405.2 [12] calls for its use on MCCR systems [18].

Some advantages of an Ada PDL are:

- Using an Ada PDL is a good transition strategy — the contractor's designers work with and think in Ada terms earlier in the development.
- An Ada PDL should be *full, compilable Ada (conforming to the ANSI/MIL-STD-1815A)*; then it can be checked by an Ada compiler on the host machine for consistency and completeness. An Ada PDL also provides an unarguable description of a design, useful to reviewers and others.
- When made a part of the documentation requirements for top-level and detailed software design documents, an Ada PDL has also proven to be a good indicator of the status and quality of a contractor's design efforts early in the development [18].

Some disadvantages of an Ada PDL are:

- Ada as a PDL has sometimes been applied in cases where the eventual implementation is not in Ada but in some traditional language. While this is technically feasible, it has been difficult to carry out in practice, and should perhaps be avoided unless the designers can demonstrate *an adequate understanding of modern software engineering* principles and implementation tradeoffs.
- Some aspects of design (e.g., performance requirements, traceability to requirements documents) are not readily captured in an Ada PDL. Structured comments known as annotations can be used to capture some of this information so that the resulting design is still compilable. Other aspects of a design (e.g., the overall design structure) are better represented with other notations such as data flow diagrams [18].

There is no standard Ada PDL, but several PDL processing tools are available. For further information, refer to [20].

See: Appendix I.

Question: Can Ada be used on MIL-STD-1750A computers?[3] On Intel 8086? On the Motorola 68000 family? Are any of these processors more appropriate to use with Ada?

Answer: Compilers exist or are close to completion for these processors. However, the mere existence of a compiler is not indicative of its quality or maturity. The quality of the code generated by the compiler (how well the mission software will run) is somewhat dependent on the capabilities of the target hardware. While Ada can, in general, run on all three processors mentioned above, the Motorola 68000 family is generally believed to provide the best support because of its large storage capability and instructions for conveniently accessing that storage.

See: 4.2.2, 5.3, 5.4.

---

[3]MIL-STD-1750A, dated 21 May 1982, "Airborne Computer Instruction Set Architecture", includes Notice 1.

Question: Should every processor in a weapon system be programmed in Ada?

Answer: Some processors have highly specialized functions (e.g., a signal processor or a Fast Fourier Transform chip). Often such processors cannot in any practical sense support all the functions typically required by a high-order language. It is inappropriate to use Ada or any other general-purpose, high-order language on such processors because the expense of building the necessary support tools is expected to be high, and the speed advantages inherent in these specialized processors may be lost by using languages for which the hardware was not intended. However, since it is likely that systems *will* use special-purpose processors in conjunction with general-purpose processors whose software has been developed in Ada, the interfaces and communication between the processors is an important item for system engineering attention.

See: 5.3, 5.4.

Question: Can Ada be used in distributed applications?

Answer: Yes. At present, distributed applications can be implemented using special-purpose executive software as was previously done with JOVIAL and FORTRAN. With continued maturity, more sophisticated methods that take advantage of Ada's tasking features should become available.

See: 5.5, 5.6.

Question: What is run-time software and why are run-time issues so important to Ada?

Answer: Run-time software provides the additional supporting functions required for executing Ada programs on a specified target computer. Each target computer and each language places requirements on the run-time software. While Ada run-time software is responsible for functions such as scheduling, parameter passing, storage allocation, and some kinds of error handling (much like other languages), there are a number of issues that may affect a program:

- The Ada run-time software accomplishes many of the functions (such as scheduling) that formerly constituted the custom "executive software" written for applications developed in CMS-2, JOVIAL, FORTRAN, etc.
- The Ada run-time software available for general purpose use does not necessarily address all the needs of a specific DoD system. Special functionality may not be present, and execution efficiency for desired features may not be optimized. For these reasons, modifications to the run-time system may be necessary.
- The Ada run-time software is largely outside the control of the system developer since it is developed, maintained, and most likely owned by the Ada compiler vendor. The run-time software is tested as part of the validation process and is much more closely mated to the compiler than has been true for other languages. Should modifications to the run-time software be desired, the program manager should be aware of the following:
  - Program funds most likely will be required.
  - Modifications potentially have validation implications.
  - Modifications can be done by either the compiler vendor or the system developer, after acquiring the necessary data rights. Should the system developer accomplish the modifications, maintenance of the modified system must be considered as a recurring cost.
- Some vendors attempt a per-copy charge for each copy of the run-time software used in an embedded weapon system.

For further details about validation policy, see [4]; for further detailed discussion of run-time systems, see [36].

See: 4.2.2.2, 5.5.

Question: What is an APSE? Is an APSE required to use Ada?

Answer: An APSE is an Ada Programming Support Environment — an integrated set of tools that supports the development and maintenance of software throughout its lifecycle. An APSE would include, but is not limited to, such services as software development and maintenance functions, project control functions, configuration management, and project documentation support.

Another frequently used term is MAPSE — Minimal Ada Programming Support Environment. The MAPSE indicates the minimal group of software tools sufficient to develop and maintain Ada software. It generally includes the compiler, linker, editor, command interpreter, debugger, and configuration manager.

While an APSE is not necessary when using Ada, and the ideal of an APSE has yet to be fully realized, the quality of individual tools can have an important effect on programmer productivity and overall program development costs; thus the tools at the MAPSE level deserve considerable program manager attention.

See: 4.3.

Question: What is the significance of the Common APSE Interface Set (CAIS) development?

Answer: The CAIS is a proposed standard set of interfaces to major operating system functions developed to facilitate the transportability of tools between Ada Program Support Environments. The current version of CAIS (CAIS 1) is defined in DoD-STD-1838; CAIS-A, a revision, is projected for standardization in 1988. At present, only a few prototype CAIS implementations exist; as such, the CAIS is not yet ready for use on *any* production programs. Additionally, there is no requirement, either technical or policy, to use CAIS. As part of the recent Nunn amendment activity within NATO countries, two near-production-quality, complete implementations of CAIS 1 are expected by mid-1988 [30].

Question: What is software portability? Why is this an important Ada advantage?

Answer: Software portability describes the extent to which computer software can be moved without modification between different computer systems. Portability has traditionally been a significant problem due to the proliferation of programming languages and their dialects. (FORTRAN is a classic example of a programming language with many dialects.) While some modification will almost always be required when moving software from system to system, Ada compares favorably to other languages because of:

- enforcement of the standard via the requirement for compiler validation,
- language features (packages and private types) that allow software developers to isolate machine dependencies from the rest of the software. Through use of these features, the scope of required modifications is localized and chances of errors diminished.

Of course, Ada must be used properly to maximize portability. In particular, portability can be compromised by:

- tradeoffs made to obtain maximum performance efficiency, and
- use of implementation-dependent language features.

Further guidance on writing transportable Ada software can be obtained from [29].

See: 4.2.1, 4.2.2.4.

Question: Can Ada be mixed with other languages?

Answer: Upgrading part of an existing system using Ada is possible when technical analysis indicates that potential problems can be solved. It is also possible to develop new systems by mixing languages, for example, calling the FORTRAN International Mathematical Software Library (IMSL) from Ada. However, not all Ada compilers currently provide the capability to mix languages [26].

See: 7.1.

Question: Are there risks in automatically translating existing systems to Ada?

Answer: There are significant risks in attempting automatic translation from any programming language to another, and this strategy is generally not recommended as a means of achieving Ada-based software. A thorough analysis of technical, cost, and lifecycle issues should be accomplished before committing to an automatic translation approach.

See: 7.4.

Question: Can database management packages and fourth-generation languages still be used?

Answer: Database management packages and fourth-generation languages, when used in their proper domain, make significant increases in productivity possible. However, they can prove very costly if used to satisfy requirements even slightly outside their domain of applicability. Some portions of mission-critical computer resource (MCCR) applications may be suitable for fourth-generation language approaches.

See: 6.

Question: Why is there interest in software reuse? Can Ada help?

Answer: Reuse is of interest because many functions in new software systems are similar, if not identical, to those in previously developed systems. Reusing existing software design and code and applying them to new development efforts has the potential to significantly reduce program development costs as well as to produce more reliable systems. Ada's standardization and some of its special features mean that Ada is especially suitable for promoting software reuse. However, Ada does not automatically make software reusable. Rather, Ada language features are available to make reuse possible.

See: 8.

2.4. Program Control

The following questions are covered in this section:

- What Ada training sources exist? What topics should be emphasized? How much time should be planned for adequate training?
- A contract has already been awarded to a contractor who has limited or no experience developing systems in Ada. What is the best strategy to use in this situation?
- How should contractors proposing Ada be evaluated?
- Will changes be required in software management methods (e.g., design walkthroughs and code inspections) and documentation? What methods can be used to measure and evaluate true progress during system and software design?
- Will Ada affect cost-estimating models?
- How productive can Ada software developers be?
- Can Ada be used with DoD-STD-2167?
- What effect does Ada have on configuration management (CM)?
- Compared to other languages, will Ada require more or less source lines of code (SLOC) for equivalent functions?

Question: What Ada training sources exist? What topics should be emphasized? How much time should be planned for adequate training?

Answer: Ada training courses are available from a wide variety of government, academic, and commercial sources. Two sources of information are the *CREASE: Catalog of Resources for Education in Ada and Software Engineering* and the *Ada-JOVIAL Newsletter*. To obtain the full benefits of using Ada, software engineering concepts should be taught as an integral part of the course. Additionally, hands-on design and programming exercises are essential. Since many software developers do not have the appropriate background in software engineering concepts, time should be allowed to teach these concepts as well as the language. For example, a FORTRAN programmer may require three weeks of initial training, plus four to six months of apprenticeship under experienced Ada personnel.

See: 9, Appendix I.

Question: A contract has already been awarded to a contractor who has limited or no experience developing systems in Ada. What is the best strategy to use in this situation?

Answer: There are many issues (for example, compiler and tool acquisition and system engineering concerns) that are detailed in other chapters of this handbook. Perhaps most important, given this situation, is to develop an internal cadre of skilled Ada personnel. The following action plan is recommended:

- Develop and execute a long-term, phased training plan that includes management personnel, application specialists, lead designers, software engineers, and testing personnel. Management training is a must. Course material must include software engineering principles.
- Structure the development schedule to contain a few noncritical Ada tasks that can be accomplished very early in the program (in parallel with requirements definition and preliminary design). The main purpose of these tasks, which could be prototypes of parts of the weapon system that will be developed, is to develop some personnel experienced in using Ada. Emphasis should be on small team efforts, with much review of the ongoing development, distribution of information, and management observation. Since Ada has been shown to change the levels of effort applicable to various phases of the lifecycle, management should not attempt to rigidly control the development of these tasks, but rather should use them as a vehicle to understand what will happen in the main development.
- Sprinkle the experienced personnel from the small Ada projects in as leaders of the mainline development [31].
- Based on the results of the early Ada tasks, adjust the software development plan (SDP) and the training plan for the main effort.
- Consider bringing in some outside experts with experience developing similar systems in Ada to evaluate the system design (hardware and software) and implementation strategy.

See: 4, 9, Appendix I.

Question: How should contractors proposing Ada be evaluated?

Answer: In general, contractor evaluation should be done in a manner similar to that used for other software developments. *A proven record of cost, schedule, quality, and application-domain expertise* is important. Additionally, an assessment of internal Ada capabilities in terms of people, tools, training, methodology, ongoing internal research and development efforts, and previous developments is necessary; a contractor attempting to do a large Ada development without any Ada record is indeed suspect. Finally, independent Ada experts can be consulted as part of the evaluation process. For additional information, see [9].

Question: Will changes be required in software management methods (e.g., design walkthroughs and code inspections) and documentation? What methods can be used to measure and evaluate progress during system and software design?

Answer: While techniques such as design walkthroughs and code inspections still apply, the levels of effort and time associated with the various phases of the software lifecycle will probably change. Experience to date indicates the need to spend more time in the definition and design phases and less time in testing and integration than was spent in previous efforts. Increased emphasis on design activities will lessen the time involved in testing and integration, since many errors will be identified early in the development [6]. Some **rules of thumb** to use regarding levels of effort are:

Phase	Other Languages	Ada
Design	40%	50% +
Code	20%	20-30%
Test and Integration	40%	20%

The documentation requirements levied on the project must reflect the architecture, design, and prototype or PDL rather than allowing the documentation requirements to dictate design. For instance, design in Ada may (and should) include both static and dynamic program structures, early prototyping, and design verification in the Ada language [34].

Question: Will Ada affect cost-estimating models?

Answer: Several of the major cost-estimating models (COCOMO, SOFTCOST, etc.) are currently under revision to reflect experience with Ada. While empirical evidence is limited, experience from initial Ada programs indicates that several cost factors will change, including productivity, definition of lines of code, and levels of effort associated with various phases of development [6, 31]. Program managers can help further refine cost-estimating models by gathering and sharing data about these topics.[4]

Question: How productive can Ada software developers be?

Answer: While only a few definitive Ada productivity studies have been conducted [10, 6, 14, 31], Ada software development efforts generally report that overall productivity is higher than for other languages. Indeed, improvements as high as a factor of two have been reported informally. However, the program manager should be aware that it is also *not* unusual for productivity at the beginning of a program to be low due to a learning curve. (A good training program would be valuable in reducing the learning curve.) Additionally, cautious use of reported data is recommended since productivity figures vary due to:

- the capabilities of people,
- the size and complexity of the system under development,
- volatility of requirements specifications,
- maturity and sophistication of the software development environment, and
- documentation requirements [5].

Question: Can Ada be used with DoD-STD-2167?[5]

Answer: Yes; however, as with any DoD standard, tailoring will be required for the particular needs of any program. DoD-STD-2167 is presently under revision (DoD-STD-2167A) to make it more compatible with Ada-based technology. A discussion of some 2167 related issues may be found in [13, 15]. Significant work in this area is also being accomplished by the Software Development Standards and Ada Working Group (SDSAWG).

See: Appendix I.

Question: What effect does Ada have on configuration management (CM)?

Answer: Disciplined CM practices are a prerequisite to effectively managing Ada software. CM for small and moderately sized Ada systems (less than 90,000 lines), as supported by traditional file oriented tools,[6] is largely indistinguishable from CM for systems written in other programming languages.

---

[4] A focal point for data collection efforts is the Cost Research Branch (ESD/ACCR) Electronic Systems Division, Hanscom AFB, Massachusetts. Additionally, the AdaJUG has recently established the Ada Data Collection Subgroup.

[5] DOD-STD-2167, "Defense System Software Development," 4 June 1985 (and subsequent revisions); DoD Handbook 281, "Defense System Software Development Handbook," 22 October 1984 (and subsequent revisions).

[6] Examples would be DEC's MMS and CMS or UNIX SCCS and Make.

However, the CM requirements for large software systems tend to be much more complex than for smaller systems, independent of programming language. This is especially true for Ada, which is intended for the design and development of large systems. Among the issues which have to be addressed are:

- multiple implementations of package bodies, and
- the growing number of interdependencies among objects as systems increase in size.

For large systems, proper system management will require support from the program library since the compiler is involved in supporting functions that previously had been managed by the environment support tools. *Since CM for large Ada systems is an area for concern,* the program manager should:

- ensure that Ada program libraries provide support for automatic recompilation, and
- evaluate the level of integration between the Ada compiler and the project-selected CM tools.

Question: Compared to other languages, will Ada require more or less source lines of code (SLOC) for equivalent functions?

Answer: There are many different views of what a source line of code is and how to use the computed values. Although the validity of SLOC has been questioned and definitive guidance on this matter is difficult, an understanding of the issues and concerns is important, since SLOC is sometimes used as an indicator of system complexity, system hardware requirements (processors and memory), and as a basis for computing system cost.

Any estimates comparing Ada and other languages are *inexact* since factors such as individual capabilities, application complexity, design and coding techniques, and language feature use will exert strong influences.

Following are *general* comparisons between Ada and other languages; it should be noted, however, that as reuse and generics become more common, the number of *original* lines (which have to be written from scratch) will decrease [34].

- Ada compared to JOVIAL: Would probably yield slightly more source lines of Ada.
- Ada compared to COBOL: Would probably yield slightly more lines of COBOL.
- Ada compared to CMS-2: Would probably yield about 1.3 times more lines of Ada.
- Ada compared to FORTRAN: Would probably yield about 1.5 times more lines of Ada. (Note that widely disparate values have been reported for FORTRAN comparisons.)

There are a number of explanations for these estimates:

- Ada is generally more specific and more complete than other languages, resulting in more lines of code. The benefit is that Ada code should be more reliable and easier to read, understand, and maintain.
- Ada is richly declarative and descriptive regarding data structures. These characteristics *increase lines of code* and also allow more errors to be caught by the compiler as compared to languages such as FORTRAN that do not possess these capabilities.

The following are **rules of thumb** to use until costing models are recalibrated:

- Use a consistent metric when computing lines of code. Counting semicolons (at end of a statement only) is a generally accepted Ada measure.
- Ada declarations should be considered as lines of code.
- Cost per line of Ada code is expected to be less than JOVIAL or FORTRAN.
- Program managers are urged to gather data. In this way, meaningful numbers for both lines of code and productivity can be developed.
- There are about 2.5 nonblank, noncomment lines per ";" in Ada, i.e., each Ada statement takes about 2.5 lines on the average [5].

2.5. Getting Help

Question: Where can one get more information about Ada?

Answer: Many sources of information are listed in the appendices to this handbook. Two sources are included here for quick reference:

- **Ada Joint Program Office (AJPO)**: Contact Virginia Castor, Director, AJPO, the Pentagon, Washington, D.C., (202) 694-0210.
- **Software Engineering Institute (SEI)**: Contact John Foreman, (412) 268-6417, or John Goodenough, (412) 268-6391.

See: 9.4, Appendix I, II, III.

3. The Need for Ada

3.1. The MCCR Software Problem and Ada's Role

Software development and post-deployment support are increasingly recognized as serious problems for mission-critical system development. Ada was designed to address these problems by:

- *reducing the number of programming languages needed so DoD effort could be focused on making one language work well; and*
- *encouraging the use of modern software development methods, thereby reducing the costs and risks of software development and facilitating improved maintenance during post-deployment support.*

Mission-critical computer resources (MCCR) are becoming increasingly vital in defense systems. Many DoD and industry studies have documented that software is assuming a greater share of system complexity and cost, and demand for quality software has been rising faster than the ability to produce it. Indeed, systems of hundreds of thousands, even millions of lines of code are becoming increasingly common, resulting in situations where computing systems are on the critical path to systems acquisition and among its leading problems.[7]

The growing importance of software cannot be overstated. Col. Donald Carter, U.S. Air Force, former Acting Deputy Undersecretary of Defense for Research and Advanced Technology, told the House Armed Services Subcommittee on Research and Development that "software is the human intelligence that is programmed into our systems. It allows advanced sensors to discriminate and track, navigation systems to follow prescribed routes, guidance systems to control trajectories, and communications systems to properly route thousands of messages. Software keeps track of the status of our forces, maintains intelligence information on enemy forces, and aids our commanders in deciding on target actions" [22].

Software engineering is the discipline in which quality (efficient, reliable, maintainable) software is produced within the constraints supplied by contractually specified acquisition, development, operational performance, and lifecycle support considerations [23]. Although software engineering methods and techniques have advanced dramatically over the last decade, these techniques have not generally been practiced in DoD software development efforts. The DoD is aware of the software cost and technology insertion problems in the development and post-deployment of MCCR,[8] and has established a software initiative consisting of the Ada program, the Software Technology for Adaptable Reliable Systems (STARS) program, and the Software Engineering Institute (SEI) to address the problem. The Ada

---

[7]To illustrate the magnitude of a 1,000,000 line software system, assume that it is printed using 8.5" x 11" paper and 50 lines per page. The resulting listings would be a stack of more than 20,000 pages. Assuming that 500 sheets is about 1.5" thick, the stack would be approximately 5 feet high. Note that the above sizing omits any consideration of code to provide the software development environment, simulation software, test software, documentation, contractually required data items, presentation and review documents, etc. [16]. This stack of paper is comparable, in terms of complexity, detail, and time required for development and maintenance, to technical orders and operating instructions used during military operations [23].

[8]The DoD moved toward language and processor standardization when there were more than 300 different computer programming languages and more than 1000 dialects in use on DoD systems, as well as perhaps hundreds [8] of target processors. This created an N*M (languages * processors) problem of enormous proportion, as the system development and post-deployment software support (PDSS) costs of this situation in terms of tools, people, training, and maintenance rapidly escalated.

language effort focuses programming development methods and tools on a single language that supports modern software engineering techniques. Ada's role as the single, common, high-order programming language for computers integral to weapon systems [12] is a major step forward in addressing DoD software development problems. Current DoD policy regarding the use of Ada is described in DoD directives 3405.1 and 3405.2 [11, 12].

3.2. Ada Benefits

Ada's major advantage is in providing the means to achieve the benefits of modern software engineering methods.

The Ada language offers potential solutions to software development problems in the areas of:

- **Code portability**: Ada makes it easier to move source code between different computers with a minimum of changes (often referred to as machine independence).
- **People portability**: With language standardization, software personnel will be more able to move from project to project with significantly less need to learn a new language. (While Ada is standardized, differences among compiler and tool implementations still exist.) Additionally, with time, there should be a larger pool of people able to work on any given project.
- **Maintainability**: Ada can reduce costs to make changes in mission-critical software.
- **Reliability**: Ada can reduce errors during development and maintenance activities by providing earlier identification of certain types of programming errors. Such capabilities will help increase system reliability.
- **Common basis for tools and methodologies**: Ada can serve as a focal point for investing in and developing high quality tools and methodologies. Previously, tool and methodology development efforts were diffused across many languages, resulting in less capable and lower quality tools.
- **Managing complexity and modularity**: Complexity has become a major factor limiting progress in the software industry. Ada helps manage complexity by explicitly supporting the process of dividing a problem into well-understood pieces whose interfaces and interactions with other software and hardware components are well defined. This support is critical because, in general, the approaches, techniques, and languages that have historically been used to build software systems do not address larger problems very well. Ada is therefore becoming a necessity for large systems to function correctly and reliably. It is the best tool available for the complexity and versatility required of DoD software [34].
- **Management visibility, more emphasis on a system view**: Ada's features encourage a more disciplined approach to defining hardware and software interfaces and making design decisions; hence, managers will have the opportunity to better understand and control their progress against schedule.
- **Productivity**: Lines of code and units of functionality produced per day (whether original or reused) are expected to increase.

While the potential advantages are great, the mere presence of Ada and supporting technology does not guarantee success. Achieving Ada's benefits means investing in training, tool development, pilot projects, and development standards as a down payment for benefits to be achieved over the entire system lifecycle.[9]

Ada's importance in supporting software engineering has been recognized outside the DoD. Commitments to use Ada in commercial developments have been made by companies such as Computer Corporation of America (CCA), Shell Oil, Boeing Commercial Airplane Company, Nippon Telephone and Telegraph (Japan), and Philips (Europe). These companies have chosen Ada because they are convinced that the technology is mature enough for their applications (which include communications, banking, and teleprocessing) and the advantages are worth pursuing. Indeed, 80% of Ada use in Europe is commercial [1]. Ada is not just a DoD language. Its success is important to private companies as well, and the DoD can share in the benefits of commercial interest and development.

[9]The system lifecycle is the span of time over which a system is in existence starting with its conception and ending with its last use. System lifecycles are usually divided into conceptual, developmental, production, and operational phases and span many years.

4. Software Production Technology

This chapter examines the tools used to build Ada software and includes:

- an introduction to software development terms and concepts used in this handbook
- a discussion of Ada compiler issues such as validation and quality indicators
- a discussion of programming support environments and additional tools
- a discussion of expectations regarding tool maturity
- a forecast of expected near-term technical progress.

4.1. Software Development Terms and Concepts

The following terms and concepts are explained in this section: host/development computer, target computer, embedded computer, source code, object code, run-time software, program library, compilers, linking, debugging, downloading, and program execution.

This section introduces generic software development terms and concepts discussed in the rest of this handbook. To focus on areas of critical importance and to provide a basis for subsequent discussions, this section concentrates on the process of transforming *source code* developed by a software engineer into *object code* which will *execute/run* on a target computer. *Readers already familiar with these items can skip to section 4.2, which discusses Ada compilers.*

Two computers are used in the process of software development:

- **Host/development computer**: The software development process typically takes place on flexible, multi-programming computers like those produced by IBM, Digital Equipment Corporation (DEC), and Data General, but it may also occur on networks of personal workstations.
- **Target computer**: The computer that runs the final mission software; defense systems examples are MIL-STD-1750A, Intel 8086, and the Motorola 68000 family.

While the host and target computers can be the same, the host computer is often different from the target computer because:

- More (and different) resources are needed to develop and compile code than to execute it. Such resources include system memory and peripherals such as disk drives.
- The target computer is *embedded* in and functions as an integral part of a weapons system. Embedded computers are typically small, special-purpose machines dedicated to specific operations or functions and therefore are not suited for developing software. Examples are computers in industrial robots, navigation systems, and process control equipment.

When the target computer is different from the host computer, the executable code must be transferred (*downloaded*) to the target computer before it can be run.

As Figure 4-1 shows, the software developer uses an interactive software tool called an *editor* to create and save (and later modify) the source code for each of the many modules (or units) which will constitute the completed system. The source code is, for the purposes of this handbook, written in a high-order language (HOL), a programming language such as Ada, COBOL, JOVIAL, or FORTRAN. HOLs enable a

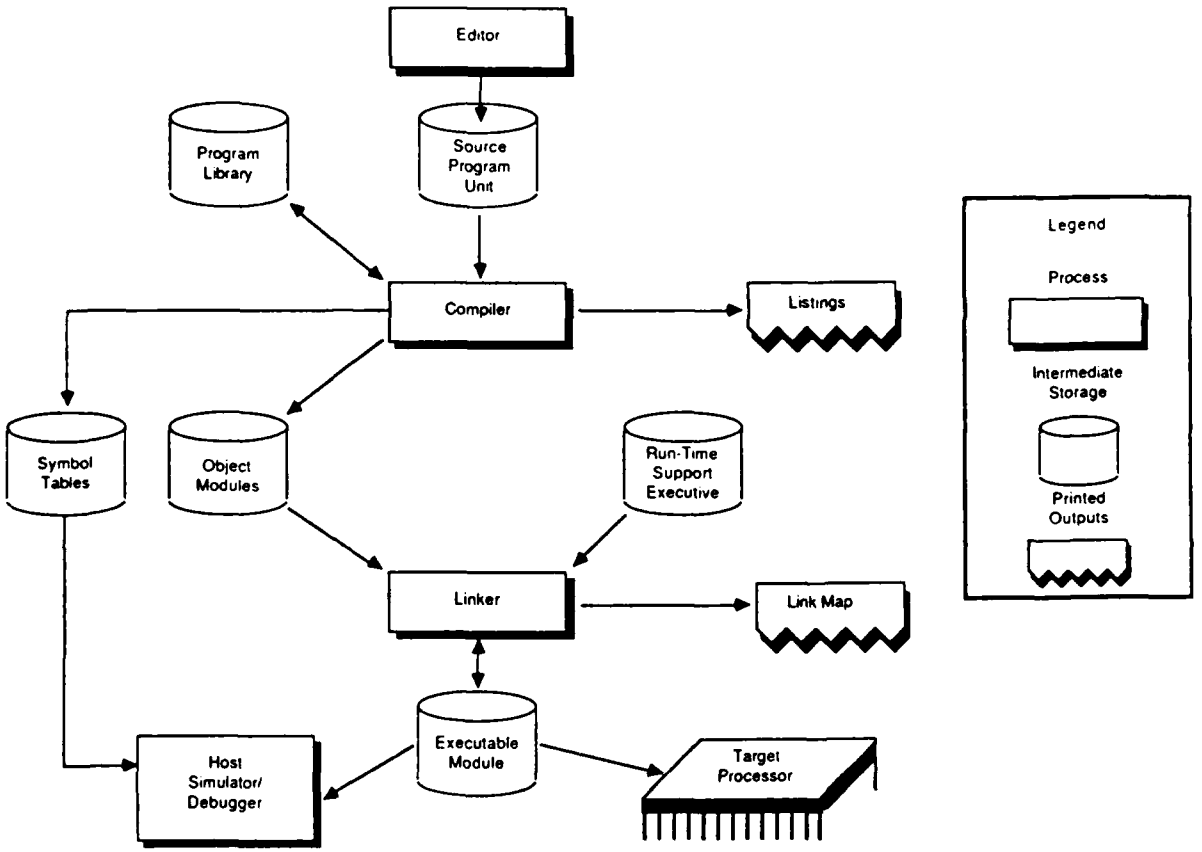


Figure 4-1: Flow diagram of tools used in the software development process.

software engineer to write in an English-like, readable form rather than in a binary machine language which is unique to each type of computer system.

After each module is created, a *compiler* checks the source code for first-level correctness by ensuring that the syntax (grammar) of the code is correct. If errors are identified, the process cannot continue, and the developer then uses the editor to correct the error(s) and resubmits the module to the compiler. If no errors are found, the compiler:

- translates the source code into *object modules* which will (eventually) run on the target computer (in DoD applications, where often the host and target computers are different, a compiler that translates source code into a machine language for other than the host computer is known as a *cross compiler*),
- produces various output listings (see paragraph 4.2.2.3), and
- produces symbol tables which are used in conjunction with debuggers.

Relationships among object modules (and in some cases, the object modules themselves) are maintained in a *program library*; libraries vary in their sophistication, depending on the requirements and capabilities of the high-order language being used.

The independently compiled object modules must be *linked* together before the program can be executed. When the application code is linked, the appropriate *run-time software* is automatically included. Run-time software provides the additional supporting functions required for executing programs on a specified target computer. Each target computer and each computer language requires its own specific run-time support library. For Ada, functions most likely handled by run-time software include scheduling, parameter passing, storage allocation, and some kinds of error handling. As part of the linking process, additional consistency checks are performed; in particular, any missing or possibly obsolete compilation units are reported. The outputs of the linking step include:

- An *executable* (ready to run) module
- Printed listings, generally called a link map, which describe the structure of the executable module.

The executable module can be executed and tested on either the host or the target machine. Usually, the host machine will continue to be used, especially during the early phases of software debugging and testing, because of the greater resources available on the host. If the host machine is not the target computer, it is possible to simulate execution of the mission software by using a *simulator/debugger*. The simulator/debugger, which is also referred to as a target simulator, runs on the host machine and mimics the timing and functional behavior of the target machine, thereby providing a good view of execution in the target domain. An additional level of problems can then be discovered *before* the mission software is downloaded to the target. If logic errors (errors in functionality and operation) are discovered, the software developer must return to the editor to begin the correction process.

After a certain confidence level is reached using the host-based simulator/debugger, the ultimate test is to download the executable mission software to the target computer and execute it there. Here, target computer-oriented debugging tools are very helpful for locating errors. Errors discovered during software execution are most expensive of all to resolve.

4.2. Ada Compilers

Several important issues should be considered when selecting an Ada compiler.

While all tools used in the software development process are important, the Ada compiler continues to attract the most attention, in part because of:

- the requirement for compiler validation, and
- compiler usability for particular applications.

These issues are addressed below.

4.2.1. Compiler Validation

> *To ensure that program-dependent dialects of Ada do not arise, the DoD requires that validated Ada compilers be used for mission-critical applications. However, not all validated compilers may be suitable for every program. Each program must select an appropriate compiler and tool set, and make improvements when necessary. Compiler modifications are acceptable if they do not change the language. Modified compilers must be revalidated, but at the discretion of the program manager.*

While Ada has been mandated as a standard by the Department of Defense, it is not enough just to have a standard — the standard must be enforced. With Ada, compliance is tested by the Ada Compiler Validation Capability (ACVC), an integrated set of tests, procedures, software tools, and documentation used to validate Ada compilers.[10] The ACVC is a dynamic test suite which has undergone continual expansion and revision since it was first introduced, and has grown from about 1700 tests to almost 3000 currently. If a compiler passes all the applicable ACVC tests, it is said to be *validated*. Based on the official AJPO list of 1 May 1987, there were 78 validated compilers targeted to at least 30 different machines. In general, only validated Ada compilers can be used for DoD applications.

Although ensuring that Ada implementations conform to the standard has a long-term benefit for the DoD, the value of validation to an individual DoD program is limited. Validation should be regarded as a necessary, but not sufficient, condition. The program manager should know that:

- Validation does not mean a compiler is free of errors — no tests can detect *all* errors in a complex software product.
- Revalidation is required periodically. The general rule is once a year, but the rules for compilers incorporated into a project baseline allow some flexibility.
- The validation process is *not* concerned with performance issues. Validation implies only a minimal level of usability, a level that may not meet the needs of a particular program or application. Consequently, each program must determine whether a validated compiler meets (or can be modified to meet) its needs. In addition to benchmarking facilities being developed by various agencies and contractors (see Appendix I for more information), a new DoD effort, the Ada Compiler Evaluation Capability (ACEC), will develop criteria and tests for evaluating compiler quality. The first release of the ACEC is scheduled for July 1988.

---

[10]The AJPO validation procedures and guidelines document [4] defines commonly used Ada validation terms; outlines the organizational structure for managing, coordinating, and directing the Ada validation process; lists steps in the process; and provides guidance to DoD program managers on the acquisition, use, and maintenance of Ada compilers. Contact the AJPO for details.

The requirement to use only validated compilers may appear to pose a risk when combined with the DoD requirement to revalidate compilers periodically. After all, if a program's compiler fails revalidation, it must be corrected, and the application code must be recompiled and completely retested, which can have cost and schedule implications. The validation procedures and guidelines document addresses these concerns by providing the following guidance:

- Any compiler placed under baseline control must be a validated compiler. Once placed in the baseline, the compiler does not have to be revalidated until the program manager finds it prudent to confirm that the compiler still conforms to the standard.
- Program managers are encouraged to use the validation tests to ensure that compiler maintenance changes and modifications have not affected the ability of the compiler to pass the tests.
- Retesting is recommended at major program baseline milestones.

In short, revalidation testing is completely under the control of the program manager and can be scheduled at times when it is least disruptive to program costs and schedule. The intent of the AJPO guidelines is to ensure that compilers continue to conform to the standard without imposing an undue burden on program managers.

4.2.2. Compiler Usability

A number of factors are indicative of compiler quality and usability. Based on evaluation, compiler improvements and/or customization may be needed for some programs.

The production quality of an Ada compiler is usually measured minimally in terms of:

- **Compile-time efficiency**: This measure includes:
  - number of source lines processed per minute
  - capacity in terms of the size systems which can be built with this compiler
  - maximum size allowed for any one unit.
- **Object-code efficiency**: Size and speed of the resulting object code, including run-time software. These factors are normally compared against assembly language for the same algorithm.
- **Compiler services**: Quality of error messages, types of error recovery, output listings, diagnostic information.
- **Support for embedded system requirements**: Ability to precisely control and interface with hardware system functions.

The following sections address these issues.

4.2.2.1. Compile-Time Efficiency

Ada compilers require more host computer resources than compilers for most other languages, in part because the compilers are required to check for more kinds of programming errors.

Ada compilation costs can be significantly reduced if a compiler has special support tools for managing separately compiled units efficiently.

Ada is intended to be used not just for embedded systems, but also for large systems — systems that contain hundreds of thousands, even millions, of lines of code. The software in such systems may be compiled hundreds of times. Some compilers are not adequate for this level of use. For example, the validation tests consist of several hundred thousand lines of code. Some compilers process all the tests in a few hours; but others can take as long as a week. The latter are clearly not usable for a large software effort, and possibly not even for a smaller embedded system application.

There are several reasons why Ada compilers will use more computing resources than compilers for other programming languages:

- **Language capability**: Ada's inherent power and complexity generally means additional processing is required by the compiler.
- **Earlier error detection**: Ada compilers can detect certain kinds of programming errors earlier than they would normally be detected in other languages. A type of error Ada will catch early is the classic "apples and oranges" problem. For example, while other languages might allow numbers representing velocity and volume to be added together, with the error eventually found during integration testing, Ada will catch such errors during compilation. Although checking for such errors has a price, a well-proven software engineering principle is that the earlier an error is detected, the cheaper it is to fix. (Of course, some errors will not be detected by a compiler for *any* language; these must still be caught during unit or integration testing.)
- **Library management**: If a compiler is not designed properly, some required error checks can consume significant resources. The key issue here is how the compiler accesses and updates an auxiliary file called the *library*. Some Ada compilers can minimize the costs of accessing and maintaining the library, and these implementations are particularly effective in their support for programming large systems. For example:
  - Checking for inconsistencies between separately compiled units[11] can contribute to compile-time inefficiency if the compiler does not use special methods to access interface information declared in separately compiled packages.
  - When an interface package is changed, some compilers cannot assess the impact of the changes. As a result, large portions of the system must be recompiled unnecessarily to ensure no inconsistencies have been introduced.
- **Optimization**: Optimization refers to the process of making the generated object code more efficient for execution while insuring that the functionality of the code remains unchanged. Optimization techniques include removing redundant code, eliminating unnecessary run-time error checking, and consolidating memory usage. Since optimization of object code is especially important in embedded systems, *some*, but not all, Ada compilers devote

[11] As part of its error checking, the compiler will ensure that a separately compiled Ada subprogram cannot be called with the wrong number of parameters. Using separately compiled units also means portions of the system can be changed without having to recompile the entire system.

considerable attention to generating high-quality code. Such optimization requires extensive machine analysis of the developed software, which may consume considerable resources. The use of host resources to produce code with increased target performance is an excellent tradeoff, after the software being optimized is logically correct. The best compiler products provide different levels of optimization so software development personnel can choose the most effective level of optimization for a particular stage of development.

The following performance requirements have been suggested for production-quality Ada compilers executing in a stand-alone configuration on a processor rated at 1 MIPS processing speed for an Ada program containing at least 200 Ada source statements [19]:

- 200 source statements per minute (elapsed time, not CPU time) when generating optimized code (i.e., code that requires no more than 30% additional target computer memory and no more than 15% additional time over equivalent software written in assembly language).
- 500 source statements per minute (elapsed time) when generating unoptimized code.
- 1000 source statements per minute (elapsed time) when checking the legality of an Ada program; no object code is generated in this case.

4.2.2.2. Object-Code Efficiency

It is technically feasible for compilers to generate code that is no more than 30% larger and 15% slower than functionally equivalent code written in assembly language. There is no reason to believe Ada compilers cannot meet these efficiency goals. However, compiler modifications may be needed to achieve them.

Production-quality Ada compilers have the potential to generate code that is more efficient than code generated for other languages.

Efficient object code can be generated by Ada compilers, although some current compilers do not yet achieve this goal. A large reason for the production of less than optimal object code at present is that vendors have not focused on building good optimizers; their primary goal has been getting a *validated* compiler to market. Just because some compilers generate inefficient code does not imply that Ada code must be inefficient; more mature compilers generate code that is as efficient as code generated for other high-order languages. In fact, some Ada features potentially allow Ada compilers to surpass the object-code efficiencies of other languages. For example:

- Certain subprogram calls can be eliminated by putting the subprogram code "inline" in place of the call, thus saving time.
- Compact representations for data can be specified when compared to the compiler's normal data layout strategy, saving space.
- Certain unnecessary data compatibility checks ordinarily performed at run time can be omitted, saving both time and space.

Not all Ada compilers support all these features; they should be required by those programs with severe object-code efficiency requirements.

The run-time system for any language can also be a source of inefficiency. Some items of concern are:

- **Loading run-time software**: With some compilers, the entire run-time software is loaded, whether it is needed or not, making the executable code unnecessarily large. Thus, the ability to include only those functions which an application needs would save space. This capability should not cause any validation concerns, but may require some additional intelligence in the compiler and the linker, as well as some reorganization of the run-time software by the vendor.
- **Run-time customization**: Significant efficiencies can be gained if portions of the run-time system are tuned to a particular application.[12]

Where severe time and space limitations exist, it is often necessary to invest in program-specific custom optimizations. With properly designed compilers, the process of creating, testing, and documenting such optimizations can be relatively inexpensive.

4.2.2.3. Additional Compiler Services

The usability of a compiler is affected by the services it provides beyond the generation of code.

Compilers differ in the extent to which they provide helpful services to software engineers. The compiler services indicated below have been helpful in many MCCR efforts [19]. For more information, see [28].

- **Formatted source code listing**: An indented listing revealing the structure and control flow of the source code. This capability is sometimes referred to as a "pretty printer"; these tools can often be tailored to allow each software development effort to establish its own formatting conventions.
- **Absolute assembly code listing**: A listing showing the absolute memory location of each machine instruction in the target computer.
- **Interleaved assembly code listing**: A listing showing the machine code generated for each Ada statement. (It is not always possible to produce such a listing for highly optimizing compilers.)
- **Set/use listing**: A listing showing where variables are modified and/or read.
- **Compiler error messages**: Explanations of programming errors found by the compiler. These messages are provided at the point of the error, in a consolidated listing, or both.

In addition, the following services are specifically Ada oriented:

- **Automatic recompilation option**: An option whereby the compiler finds and automatically recompiles all modules requiring recompilation.
- **Recompilation analyzer**: A capability (in some cases, a separate tool) that analyzes the structure of software and indicates how much recompilation will be caused by various changes.

---

[12]For example, part of the run-time system might be replaced with specialized code that is particularly efficient when the language is used in certain restricted ways. Some target computers, for instance, have special instructions for manipulating text strings shorter than a certain size. The run-time software for string handling must, in principle, deal with strings of any size; but if the application code never uses strings above a certain size, the general-purpose run-time software can be replaced with more efficient, specialized code.

4.2.2.4. Special Embedded System Requirements

Ada compilers are now being built to address the special needs of embedded systems.

Ada was intended for DoD embedded system applications, some of which impose special requirements on compilers. For example, embedded systems typically need to:

- process hardware interrupts
- place data in particular memory locations, e.g., to deal with memory-mapped I/O or to ensure that overlays will work properly
- specify storage-efficient representations of data
- implement code in ROM
- access specialized low-level machine instructions, e.g., special instructions that compute trigonometric functions or that allow memory checks (referred to as a built-in test — BIT) to be run when a system would otherwise be idle
- accomplish required actions at precisely specified times.

Ada implementations have not, in general, addressed these specialized needs because of the pressures and demands of achieving validation and because these features are not currently required as part of the validation process.[13] However, validated compilers that meet the special needs of embedded and mission-critical systems are now beginning to appear.

4.2.3. Compilers: Action Plan

The following suggestions provide guidance for compiler acquisition:

- **Host machine resources**: Plan to support larger host machine resources (more memory and computing power) than for other languages.
- **Compiler evaluation**: Specify additional criteria and tests a validated compiler must satisfy to meet program-specific needs.
- **Compiler services**: Ensure that needed user support services are provided by the compiler.
- **Recompilation costs**: Assess vendor support for reducing the costs of compiling (and recompiling) large software systems.
- **Revalidation**: Balance the costs of revalidation (including recompiling and retesting all mission software) with the advantages a new compiler version may have in terms of reliability (fewer bugs) and improved object-code quality.

[13]Several of the listed capabilities are referred to as "Chapter 13" because they are included in Chapter 13 of the standard.

4.3. Programming Support Environments

4.3.1. Overall Status

Because software support environments have not yet reached their full potential for any programming language, program managers should concentrate on tool functionality and compatibility.

The function of an Ada Program Support Environment (APSE) is to support the development and maintenance of large-scale software systems written in Ada. The APSE should provide support across the development and maintenance cycle to include specification development, test generation and testing, program management, coding, debugging, and configuration management [25]. The goals of an APSE complement the goals of Ada, and include:

- common tools and interfaces,
- tool integration, and
- software transportability.

While encouraging efforts are under development, systems available now for *any* programming language do not maintain development history in a way that links requirements, design specifications, code documentation, source code, compiled code, problem reports, code changes, tests, and test-run results. Program managers should primarily concentrate on:

- tool functionality
- operational compatibility between tools (including interfaces)

The following sections provide action plans for both required and optional development tools.

4.3.2. Required Tools: Action Plan

As a first objective, a program manager should ensure that a good basic tool set is used. In addition to a compiler, the minimally required tools include:

- **Editor**: This is an interactive tool for creating documentation and source code.
- **Debugger**: While development can be accomplished (sometimes with great difficulty) without debuggers, symbolic debuggers[14] for Ada are fast becoming essential development and productivity tools. The debugger and compiler work as a complementary pair, where both tools share common data structures known as symbol tables (refer back to Figure 4-1) and are therefore best purchased from the same vendor.
- **Linker**: The linker joins together the various object modules and the run-time software into an executable module which will run on the target machine. As part of this process, it may make adjustments to the code based on the intricacies of the target processor.
- **Configuration manager**: This tool is used to achieve, at a minimum, version control of both documentation and code.
- **Target simulator**: While development can be accomplished without simulators, they are becoming essential development and productivity tools. They allow more development work to be done on the host machine, as opposed to repeated and often time-consuming downloading to the target.
- **Downloader**: This tool is used to download the executable code from the host machine to the target processor. Sometimes downloader software is supplied by target computer vendors.

4.3.3. Optional Tools: Action Plan

There are many other tools which are useful, but not essential, to the software development process. Several of these are briefly explained in Table 4-1.

Before decisions are made to acquire and develop additional tools, consideration should be given to:

- **Size of the software development effort**: Small development efforts — for example, on a personal computer (PC) — may not warrant additional tools. However, in the case of large, multi-team efforts, possibly involving geographically distant participants (such as the Army Light Helicopter Experimental (LHX) and the Air Force Advanced Tactical Fighter (ATF) programs), tools and capabilities beyond those described in Table 4-1 will be required, as will significant management attention to the development process.
- **Documentation requirements**: User guides and online documentation should be evaluated for availability and quality.
- **Functionality, interfaces, and compatibility**: New tools should be evaluated for compatibility with an organization's existing tools and the Ada language. An example is design tools: Are they compatible with Ada's assumptions about software organization such as packages? When public domain tools are being considered, the degree of modification required must be determined.

[14]Symbolic debuggers allow for execution of software under programmer control. They provide methods for stopping software execution at specific points and for examining and modifying data locations using source code names.

Tool	Purpose
Source-code formatter	Also known as "pretty printers," these tools format a source file using standard, predefined coding conventions, thereby ensuring a standard visual format for code. Since formatting conventions may differ between software development efforts, the formatter tool should allow an organization to specify the conventions to be used.
Object-code analyzer	Analyzes execution paths in object code. This information can aid in testing and determining which code could be further optimized.
Static analyzer	Provides characteristic information about software such as number of lines of code, number of comments, control flow complexity, and measures of coding style. A compiler may provide this information, eliminating the need for this tool.
Dynamic analyzer	Monitors and reports the execution behavior of code so its performance can be tuned.
Source-code cross referencer	Gives a cross-referenced table specifying where symbols in the software are declared and used, and how they are used. Some compilers provide this information directly. A more advanced tool allows a user to "browse" on-line, moving automatically from the place where a name is used to the place where it is defined.
Syntax-directed editors	Assist in creating source code by analyzing and checking for many types of errors as the source code is entered. The advantage of these advanced editors (also known as language-sensitive editors) is identifying certain types of errors early, without having to process the code through a compiler.
DoD-STD-2167 Documentation generators	Support the generation of documentation according to the formats of DoD-STD-2167 data item descriptions (DIDs). Often employed in conjunction with program design language (PDL) tools and/or graphical programming tools.
Graphical programming tools	Allow software engineers to visualize the design process and the resulting system design as it develops. These tools are relatively new, but progress in this area has been encouraging.
PDL processor	Supports the processing and documentation of software designs. (Given the requirements of DoD directive 3405.2 [12], PDL processors could be considered a required tool.)
Test manager	Helps organize and execute software tests, including comparing test results from previous runs.
Module manager	Accepts directions about how to build a system baseline; determines which modules in a system have been changed and which modules are potentially affected by the changes.

Table 4-1: Optional Tools

4.4. Introducing New Capabilities

Software development tools should reach a level of maturity before being used extensively in production.

Tools are valuable aids for increasing productivity and improving software quality. Software tools (including compilers) have a development lifecycle very similar to the lifecycle of mission software:

- requirements development
- design
- product development
- in-house testing
- Beta testing[15]
- initial delivery
- continued refinement
- production quality
- maturity
- obsolescence.

The maturation process from initial delivery to production quality is a key period and can take significant time. Immature tools (just out of development) can have a negative impact on productivity if software developers must concurrently debug new applications software, new tools, and new hardware [24]. Often new tools are abandoned because they are not understood or they fall short of expectations. Program managers should allow time in program schedules for learning how to use a new tool, developing proficiency, and discovering and correcting problems and limitations.

4.5. Forecast for the Future

Initial Ada compilers primarily focused on adhering to the language standard and/or establishing a market presence. Second-generation, production-quality compilers are beginning to appear, and other tools are receiving significant attention.

The DoD commitment to a standardized language has resulted in a concentrated effort to produce Ada compilers and tools. This differs significantly from the historical dilution of effort which existed when many languages were being used on defense systems. Expectations about the near future include:

[15]Beta testing is a limited, constrained test/evaluation performed very close to final completion of a product for the purposes of getting user reaction and finding additional bugs. It is usually performed outside the development group.

- **Increased compiler availability**: In 1986, there was a significant increase in the availability of compilers. While most of these compilers were still targeted for host machines, several were for workstations and PCs. 1986 also marked the introduction of compilers validated for embedded targets. This positive trend indicates that vendors have solved many of the technical problems associated with validation and have also made significant progress in rehosting and retargetting.
- **Production-quality improvement**: Significant improvements are evident in second-generation compilers, and engineering refinements should continue as competition drives the marketplace and the rate of change in the validation suite slows down. Improvements should occur in:
  - compiler capacity
  - code optimization (and therefore operational performance)
  - library management.
- **Embedded processors**: More compilers will be targeted to the resource-constrained, embedded-systems market, and efforts will address customizing run-time systems, as well as target debugging.
- **Tool sets**: Continued product development and improvement are expected in:
  - tools which support various methodologies and automate parts of the DoD-STD-2167 documentation process
  - tools and techniques for graphical systems representation
  - requirements and design traceability
  - target machine-oriented tools.
- **Development of quality metrics and evaluation techniques**: Many government and industrial organizations are developing compiler benchmarking capabilities. Specifically, the Evaluation and Validation (E&V) team (see Appendix I) and the Software Engineering Institute [37] are developing well-researched techniques to increase the sophistication of compiler and tool evaluation.
- **Continued predominance of the commercial sector**: Ada is not exclusively a DoD language. While DoD-sponsored/contracted efforts *have* provided motivation and seeds for the commercial sector, most Ada compiler successes have resulted from the efforts of commercial software houses targeting the DoD marketplace.

5. Ada Maturity and Applicability

This chapter examines the maturity and applicability of Ada for various application domains and system designs. The following topics are examined:

- a model of technology development and insertion
- scenarios for Ada use
- Ada and system engineering
- Ada and embedded processors
- Ada for real-time systems
- Ada for distributed systems.

5.1. A Model of Technology Development and Insertion

> *Any new technology goes through a readily identifiable process of development and maturation. The technology supporting Ada is following this same process.*

The process of technology development in both hardware and software generally includes the following phases:

- identification of problem or need
- development and insertion of initial product solution
- iteration:
  - use
  - feedback
  - refinement
- maturity.

Ada is following this model and is now in the early phases of the product use and refinement iteration.

Time to maturity will vary depending on the requirements of a particular application domain. Application domains are categorized based on:

- type of computers used as the target processors
- processing requirements (real-time, etc.)
- resource constraints (memory, peripheral equipment, etc.)
- physical environments (ground based, airborne, spaceborne, etc.).

5.2. Scenarios for Ada Use

Ada can be used today for applications that are not time critical and that run on machines with medium to large memory capacity. For time-critical applications on resource-constrained machines, special care is required in selecting appropriate compilers and tools.

Given the wide spectrum of DoD applications and the wide variety of processors in use, any evaluation of Ada applicability must address:

- requirements of the application domain
- maturity of software technologies
- capabilities of candidate target processors
- maturity and capability of development organizations.

The following paragraphs briefly survey the current situation and answer the question, "How and where can Ada be used?"

- **Low Risk**: Ada is viable for use today in applications that are not time critical and run on machines with medium to large memory capacity. (This includes such processors as DEC VAX, Data General MV series, and some of the newer 32 bit micro processors.) Applications include software tools, some simulations, and ground-based mission applications. This evaluation is based on current compilers and the successful use of Ada in these domains. (For further information, refer to Appendix II.)
- **Medium Risk**:
  - Large command, control, communications, and intelligence (C3I) projects involving complex interfaces and distributed processing/databasing need new methodologies and design/documentation approaches. Interfacing to commercial software packages can also be challenging.
  - Time-critical, ground-based applications running on machines with medium to large memory capacity going into *immediate* development.
  - Time-critical, airborne/spaceborne applications going into full-scale engineering development (FSED) in the next 2 to 4 years. Although the first compilers targeted to military embedded processors have appeared, more time is required for these tools to mature, since this is a more difficult application domain which uses resource constrained machines. This situation can be made somewhat less risky by selecting target processors which provide more Ada support, as indicated in Table 5-1 in Section 5.4.

  In any of these medium-risk scenarios, program management must make judgments about the rate of maturation of Ada technology compared to the schedule of the actual program. Consideration might be given to including (and funding) risk reduction and/or technology insertion phases in the program. (Areas of concern regarding compilers and run-time systems were discussed in Section 4.2.)
- **High Risk**: Immediate use (e.g., developing code) on time-critical airborne/spaceborne applications using embedded avionics processors, unless the possible problems and delays inherent in maturing the code generators and run-time systems in parallel with developing the application can be tolerated.

5.3. Ada and System Engineering

The DoD emphasis on Ada reflects the increasing importance of software in making system engineering tradeoffs.

Successful adoption of Ada depends not only on the maturity of compilers and software tools, but also on system engineering analysis of the program schedule, hardware choices, performance requirements, and supportability issues; these should be mapped against expected Ada maturation. The program manager *must* insure that software tools are available for the processors selected; joint selection of the target computer and support software is the optimal approach. In the following subsections, various system engineering situations are examined, and possible action plans suggested.

5.3.1. Absence of Compiler: Action Plan

If no Ada compiler exists for the selected target computer:

1. Select another target computer for which Ada compilers already exist.
2. If program cost and schedule allow, have the compiler developed. If the manufacturer of the target processor provides a tool set (linker, loader, assembler, etc.), determine what extensions or additional capabilities may be required. Guidance regarding cost and schedule for compiler development was discussed in Section 2.2. In addition, the program manager should consider the time required for new or modified tools to reach an acceptable level of maturity, as discussed in Section 4.4.

5.3.2. Nonstandard Memory Architectures: Action Plan

While innovative approaches to processor and memory configurations are always encouraged, prior to adoption:

1. Consider potential difficulties in hardware evolution as the system goes through its operational lifecycle.
2. Evaluate the impact on required software support. The costs of supporting unique or special versions of software development tools for nonstandard memory configurations over the system lifecycle can be considerable. The program manager should determine what special tools (if any) will be necessary and what modifications to existing tools will be required, and balance this against the advantages of the proposed hardware configuration.

5.3.3. Extended Memory Requirements: Action Plan

If a memory configuration greater than 128K (64K words instruction, 64K words data) is to be used, some 16 bit computers which do not have a large linear address space will require special support tools and run-time software. This is true regardless of the programming language.

5.3.4. Immature Compilers: Action Plan

Immature though validated compilers can fail to successfully compile some possibly complex but legal Ada code. If this happens, consider these alternatives [5]:

1. After sound engineering analysis, avoid the use of selected Ada features which cause or are likely to cause difficulty.
2. Obtain a more mature compiler.
3. Work with the vendor to improve the compiler's robustness.

5.3.5. Object-Code Efficiency: Action Plan

Where the quality of code generated by an Ada compiler might be an issue, there are several alternatives:

1. Obtain a compiler that produces better quality code for the selected processor.
2. Alternatively, fund modifications to an existing compiler or the development of a new compiler which produces better quality code for the selected processor. Guidance regarding cost and schedule for compiler development was discussed in Section 2.2. Work with the compiler vendor to provide a series of benchmarks specific to the application which the compiler *must handle with adequate performance* as a basis for accepting the modified compiler [18].
3. After sound engineering analysis, avoid using language features that the compiler implements poorly.
4. Add resources (such as additional memory) to the selected target processor. However, adding memory *may* require modifications and enhancements to the existing support tool set (see 5.3.3), and may impact weight, cooling, power, and other requirements.
5. Upgrade to a more powerful processor. While this is in principle the best solution for making a more evolvable system, there may be issues regarding processor Mil qualification and supportability after the system has been deployed.
6. Add additional processor(s) and decompose the software onto the processors. Concerns here are the additional overhead and complexity of the communication protocols (more software) which will be needed for the processors to exchange information. This processing load is difficult to quantify and is further affected by the types of busses used to connect the processors. Adding additional processors may also cause weight, cooling, power, and reliability problems.
7. Selectively use assembly language. Based on sound engineering analysis of system throughput bottlenecks, code parts of the system in assembly language. Note that the compiler in use *must* support either machine code inserts or *pragma*[16] INTERFACE for assembly language. (For additional information about mixing languages, see Section 7.1.)

[16]A pragma is an instruction to the compiler to perform some special action such as compiler optimization and set-up interfaces to software written in other languages.

5.4. Ada and Embedded Processors

It is generally not cost effective to build or use any high-order language, including Ada, on processors with unusual architectures, highly restricted instruction sets, or very specialized purposes.

Using *any* HOL (high-order language), including Ada, may not be cost effective or even possible for certain classes of processors, since all processors have varied capabilities. Some processors, such as those designed for use in artificial intelligence (AI) or signal processing applications, are designed and optimized for single purposes, while others are intended for general-purpose use. This orientation directly affects what programming language(s) can be used on a particular processor. Some recently developed processors (which possess large storage capabilities and instructions for conveniently accessing this storage) provide a more attractive target for high-order languages (including Ada) than do older, more resource-constrained processors.

Table 5-1 assesses the applicability of Ada on various processors based on the following factors:

- **Tool development**: Effort and cost required in developing and maintaining compilers and support tools.
- **Quality of the resulting product**: Specifically, what kind of code is generated? After the code is generated, will additional application and project-specific optimization be needed?
- **Affect on software developers**: Does the underlying hardware help or hinder the application software developer? Are there attributes of the target processor which add complexity, lower productivity, and increase the possibility of error?

Assessment	Description and Example Processors
Applicable	General-purpose 32 bit processors characterized by large linear address space and rich instruction set. **Examples**: DEC MicroVax, Intel 80386, Motorola 68020, ROLM Hawk-32, National 32032, Vector processors.
Applicable with Minor Reservations	Newer, general-purpose 16 bit processors and resource constrained 32 bit processors. Extended memory and distributed machine applications might provide minor problems. For RISC machines, there are no fundamental problems, but some Ada features may cause large/inefficient code structures. **Examples**: Intel 80286, Intel 80186, Motorola 68010, Motorola 68000, National 32016, AN/UYK-43, RISC processors
Applicable with Reservations	Older, general-purpose 16 bit processors, usually characterized by hardware memory constraints. Expected problems are extended memory and distributed machine applications. **Examples**: Intel 8086, MIL-STD-1750A, AN/UYK-44, AN/AYK-14, 1666B, Z8000
Limited Applicability	Special-purpose machines. Implementations of Ada compilers and support tools have not been accomplished for these processors, and at present these efforts are believed to be high risk, expensive, and generally beyond the current state of the practice. Next-generation processors of this class may show considerable improvements due to the expected addition of capabilities usually associated with general-purpose machines. **Examples**: Digital signal processors, array processors
Applicability Unknown	Not enough information exists to classify these processors. **Examples**: Application Specific Integrated Circuits (ASIC)

Table 5-1: Ada on Various Processors

5.5. Ada for Real-Time Systems

Most of Ada's benefits can be achieved for real-time systems even if Ada's tasking features are not used.

Concerns have been raised about Ada's usefulness in programming real-time systems.[17] These concerns fall into two areas:

- **Efficiency**
  - Some Ada implementations do not handle interrupts with acceptable efficiency.
  - Some Ada implementations take too long to switch from running one task to another. (*Tasks* are Ada's way of achieving concurrent processing.)
  - Code generated by some Ada compilers is less efficient in time and space than that generated by compilers for other languages.
- **Timing Control**
  - When a task needs to execute at a periodic rate, it appears to be difficult to control the rate with sufficient accuracy.
  - Ada tasks are scheduled for execution under control of run-time software (a *scheduler* or *executive*) written by the compiler vendor. If this software does not meet the needs of a particular real-time application, it may be difficult or costly to modify unless this possibility has been taken into account from the start.
  - It can be difficult to analyze the timing behavior of a system of Ada tasks, and the behavior can depend significantly on run-time software that is not under the direct control of application programmers.

These concerns generally apply to current implementations of compilers and run-time systems, not to the language, and do not mean that Ada cannot be used for real-time systems. Real-time systems have been written in FORTRAN and JOVIAL, languages which do not support concurrent processing directly. Just because Ada has features for concurrent processing does not mean these functions have to be used for real-time systems. *Most of Ada's benefits can be achieved for real-time systems even if Ada's tasking features are not used.* If Ada tasks are not used, the following considerations are important:

- Compiler implementations which do not automatically load the unneeded run-time support for tasking will save storage space.
- The need to use an existing real-time executive or write a special-purpose executive *must* be included as part of any project plan.

Not all Ada implementations are inefficient in their support for tasking. For example, some implementations can handle interrupts with an efficiency equal to assembly code. (Usually this approach requires the use of pragmas.) Some special-purpose machines support Ada tasking particularly well; these machines can switch between tasks with great efficiency.[18] The efficiency of code can be expected

---

[17] Real-time usually refers to fast-response computer processing in which the computer controls, directs, or influences the outcome of an activity or process in response to the data it receives from the activity or process. Examples of real-time systems are process control, target acquisition and tracking, and computer-aided navigation.

[18] A machine being built for telecommunications and avionics use by Ericsson in Sweden supports Ada tasking with considerable efficiency.

to improve considerably as Ada compilers mature. In short, concerns about inefficiency will prove less well founded as Ada implementations mature.

The concerns about timing control are less easily addressed. The near-term answer is simply to avoid using Ada tasks for systems that have critical real-time requirements. This means continuing to use the traditional cyclic executive approach.[19] Although cyclic executives are inherently harder to maintain than other forms of real-time software, they have significant efficiency advantages and can be written effectively in Ada. The cyclic approach can be made simpler by using Ada tasks just for interrupt handling and background processing; this use of Ada tasking can give Ada a significant advantage over other languages.

5.5.1. Real-Time Systems: Action Plan

To ensure the successful use of Ada today for a real-time system:

1. Check that the proposed compiler supports interrupt handling efficiently. If it does not, interrupt handling (which is a small portion of a system's overall code) should be done in assembly language.
2. Use a limited number of Ada tasks in the Ada code. Interrupt handling and background processing are good choices because their use can simplify the design of a cyclic executive and increase its maintainability.

5.6. Ada for Distributed Systems

Ada should provide improved maintainability and reduced long-term costs when used for developing distributed systems.

Although no currently approved DoD language was designed specifically for developing distributed systems, Ada can be used in this domain. One way to use Ada for distributed systems is to write independent Ada programs that execute on each node of the system. The software resident in the various nodes would communicate by a message-handling protocol implemented outside the Ada language. Distributed systems have been written successfully this way using languages other than Ada. Ada can be used the same way. The advantages of using Ada instead of some other language are improved maintainability and reduced long-term costs.

New ways of using Ada for distributed systems are currently the subject of numerous independent research and development (IR&D) efforts. These approaches treat the distributed system as a single Ada program whose parts execute on different nodes of the system. Specialized support software is needed if Ada is to be used this way. Although some promising research results have been obtained, the feasibility of this approach for use in production programs remains to be demonstrated.

---

[19] A *cyclic executive* is a method of organizing real-time software so actions are executed periodically (cyclically) at a fixed rate predetermined by the system designer.

6. Using Special-Purpose Languages

Currently, Ada is better for some applications than for others.

Special-purpose (or problem-oriented) programming languages have historically been developed to support highly specialized applications. As such, there is a large, mature base of software in these languages and application areas which may take several years for Ada to replace. The languages (and application areas) considered in this section are:

- **ATLAS**: for automatic test equipment applications
- **SIMSCRIPT** or **GPSS**: for simulation applications
- **LISP**: for artificial intelligence applications
- **"fourth-generation" languages**: for database and interactive applications.

6.1. ATLAS

DoD policy specifies that Ada is preferred (but not required) as the language to be used for hardware unit-under-test equipment [12], although ATLAS is also an acceptable and often appropriate choice.

ATLAS is a DoD-approved language [11] for programming automatic test equipment (ATE). It contains special features that allow the direct expression of wave forms, timing requirements, and stimulation patterns, all of which are critical to the development of this type of software.

Although the functionality of ATLAS could be supported in Ada (by writing subroutines in Ada that have functional capabilities similar to those provided by ATLAS commands), much code provided directly by ATLAS would have to be specially written if Ada were used. While much of this software could be reused in subsequent applications, the initial development cost would be significantly higher if the required software were not already available in Ada. In addition, software written in ATLAS may be more readable by test engineers than equivalent software written in Ada, simply because ATLAS has special language constructs that are particularly suited to expressing ATE algorithms. On the other hand, Ada's superior capabilities for giving large software systems an understandable structure make it a preferable choice for large, complex ATE projects.

6.2. SIMSCRIPT and GPSS

For simulation programming, it may be preferable to use one of the special-purpose simulation languages; nevertheless, simulation software is being written effectively in Ada.

SIMSCRIPT and GPSS are examples of special-purpose languages for programming simulations. For example, these languages allow a software developer to sample from various probability distributions, specify when certain events of interest to the simulation are to occur, and indicate what data are to be gathered when the events occur. They also have special features that help in reporting results.

Effective simulation software can be written in Ada. Since Ada has no language constructs or predefined subroutines specially designed for simulation programming, simulation developers will initially

have to write more code to provide the same facilities in Ada. Ada software can be written to support simulation programming, and many of Ada's initial applications are in the simulation arena (see Appendix II). Once written, these routines can be used by future simulation development efforts. But a special-purpose simulation programming language may be more appropriate until the extra Ada routines are written.

6.3. LISP

Once an artificial intelligence (AI) technique is well understood and is to be engineered into an application, it may be appropriate to use Ada as the language for implementing the operational application. For AI research, LISP is more appropriate.

LISP is the language of choice for artificial intelligence (AI) research. Like Ada (and unlike ATLAS, SIMSCRIPT, or GPSS), LISP is a general-purpose language; but LISP is generally used by small groups working on prototypes. LISP programming is typically done in a rich programming support environment; the programming support tools (which are written in LISP) are a strong reason for continued interest in LISP.

LISP is used for AI research applications because LISP software is easy to change. Ease of change is important in AI research, where the objective is usually to discover how to make a computer perform a task that is not well understood. LISP offers a high degree of flexibility to small teams of skilled developers, but that very flexibility may be inappropriate for a large team of programmers facing a significant engineering activity.

Ada supports a different view of software development, which says that changes can be hard to implement correctly and should be done using a language that helps detect inconsistencies in a changed system, and supports administrative control over the change process. This viewpoint is appropriate for long-lived systems, which may have several versions to be managed and maintained.

There is no current DoD directive that requires the use of Ada for AI research. The issue is less clear, however, when moving from research to operational applications. If the result of some AI research is to be used in an operational system, LISP may no longer be the most appropriate language. For example, making software easy to change has a price — LISP software is typically less efficient in the use of time and space than is Ada. In addition, because LISP is easy to change, it is easy to change incorrectly; such errors are usually easy to fix during the research phase but difficult to fix when a system is widely distributed and in operational use.

Depending on a sound engineering study of an application, LISP may be too inefficient for operational use. If so, the information developed during the research phase can be used to design a more efficient (albeit less easily changed) system that can be used operationally. In such cases, it will probably be better to implement the operational system using Ada. On the other hand, LISP may continue to be an appropriate choice for an operational system if:

- time and space efficiency are not critical or the target computer is specially designed to execute LISP code efficiently; and
- functional changes will be required frequently (e.g., at least monthly), but the nature of the required changes cannot be predicted easily; and

- sufficient safeguards are taken to ensure that changes are made correctly or to minimize the effect of an incorrect change.

6.4. Fourth-Generation Languages

> *Fourth-generation programming languages are cost effective when used appropriately. Their main drawbacks are limited domain of usability (they cannot satisfy the needs of some applications), slower performance, and possible dependence on a single vendor.*

The term "fourth-generation languages" is actually a misnomer because these languages are not direct descendants of, and not necessarily improvements over, third-generation, general-purpose languages. They might better be called program generators or application-specific tools. Nevertheless, these application-specific languages are designed for a limited domain, and the term covers, among others, application-specific database languages and nonprocedural languages.

If an application is well matched to a fourth-generation language, the cost of realizing the application can be significantly less expensive than programming it in a general-purpose language such as Ada. For example, spread sheets are routinely used to accomplish in minutes tasks that would require hours in a general-purpose language like Ada.

While fourth-generation languages offer significant advantages in productivity and rapid prototyping, there are some significant disadvantages to consider:

- **Limited domain of applicability**: Fourth-generation languages are quite effective when used for appropriate applications, but the area of appropriate use is bounded by brick walls — if some part of an application lies outside the domain of a fourth-generation language, the language may not be usable with reasonable efficiency or at all. One might begin using a fourth-generation language with great success, only to discover later that certain requirements cannot be met successfully and a completely new approach is required.
- **Performance**: An application using a fourth-generation language usually runs much slower than an equivalent solution written in a high-order language. This performance degradation may be an acceptable tradeoff in some applications, given the increase in programming productivity, but fourth-generation languages are usually not well suited to applications that stretch the performance capabilities of a system.
- **Vendor dependence or lack of control**: Fourth-generation languages tend to be commercial products and are thus highly susceptible to the pressures of the marketplace. If an application is meant to have an extended lifecycle, the product may change and become incompatible or even disappear [18].

Rule of thumb: Chances of success with a fourth-generation language are good if the requirements fall squarely within the domain of the particular fourth-generation language and if successful previous examples exist. But if these criteria are not met, there is considerable risk that the effort will result in failure.

7. Mixing Ada with Other Languages

Mixing Ada with other languages is possible, provided consideration is given to technical and management issues. On the other hand, "automatic" translation of code written in other languages should be approached cautiously.

The question of mixing languages is important to managers of both new developments and existing operational systems (post-deployment software support). Among the questions of interest are:

- Should Ada be used in the next upgrade?
- Is it appropriate to write a system partly in Ada and partly in some other language, thereby creating a hybrid system?

The following criteria can be used to answer these questions:

- Can the Ada code interface with code written in another language? See Section 7.1.
- Is the replaced portion relatively independent of the rest of the system? See Section 7.2.
- Is much of the system being replaced? See Section 7.3.
- If a hybrid system seems inappropriate, should the system be rewritten entirely in Ada, even though large portions of code would not otherwise have to be changed? See Section 7.4.

7.1. Interfacing Ada Code with Other Languages

Among the issues to be considered in attempting to interface Ada with other languages are:

- Can the Ada code call subroutines written in other languages? Alternatively, does the new Ada code have to be called from the rest of the system? For example, can a COBOL or JOVIAL compiler call Ada routines?
- What kind of data have to be processed by both the Ada code and the rest of the system? Can the data be represented in a way that is acceptable to both languages?
- How much redundant or unused run-time support software will be present in the completed system? Can the run-time for the languages be made to co-exist efficiently?

The following subsections present further details.

7.1.1. Subroutines Not Written in Ada

Efficiently interfacing Ada with another language requires the ability to call subroutines written in other languages.

Subroutines written in COBOL, JOVIAL, assembly, or some other language can be called from the Ada software *if* an Ada compiler supports the *pragma* INTERFACE[20] for those languages. If this pragma is not supported for the language and compiler[21] of interest, the cost and schedule implications of obtaining support should be determined. Similarly, if the existing code must call Ada routines, modifications to the compiler for the other language may also be necessary.

---

[20] A pragma is an instruction to the compiler to perform some special action such as compiler optimization and set-up interfaces to software written in other languages.

[21] The interface is created to a *specific* FORTRAN compiler, not to just any FORTRAN compiler.

7.1.2. Compatible Data Representation

Compatibility of data representation is a critical although often overlooked factor in interfacing with another language. Incompatibility can lead to inefficient code.

Another requirement for implementing a hybrid system is assuring the compatibility of data representation between Ada and the other language.[22] The data interface between the languages must be specified and controlled, just as it is between processors and busses. Key questions program managers must answer include:

- What data will be interchanged between the Ada and non-Ada portions of the system?
- Will the storage layout for these data be the same for both languages? If not, can it be made the same?[23]
- If the storage layout cannot be made identical, how will efficiency be affected?

7.1.3. Redundant Run-Time Support

Using two languages requires modifying a system so space is not wasted by functionally redundant support software.

Run-time support for a language is the code needed to implement its complex features. For a language like COBOL, FORTRAN, or Ada, the run-time code needed to support input/output (I/O) operations can be significant. If a system contains code written in two or more languages, the run-time support code for both languages will generally be needed. For example, suppose a system is written in COBOL. COBOL has extensive I/O support facilities. Suppose the unmodified portion of the original system continues to use COBOL's I/O facilities, and the replaced portion uses Ada's I/O facilities. Run-time support code will be needed for both the Ada and COBOL portions of the system, which can be unnecessarily expensive in space. If, instead, the modifications were designed so that all I/O processing were done by the COBOL code, then Ada's I/O run-time support code need not be loaded. The savings in space could be significant in a resource-limited system.

Although it may not be necessary to load Ada's run-time support for I/O, some less mature compilers may load it anyway. The extent to which a compiler loads only the needed run-time support software is an important factor in deciding whether it is feasible or cost effective to write just a portion of a system in Ada. Only a detailed analysis will provide answers to these important factors.

---

[22]For example, arrays in FORTRAN are typically stored differently from arrays in Ada. So, although a FORTRAN matrix inversion routine may be called from Ada, the FORTRAN language will view the matrix as though its data are arranged differently from the Ada view and will produce incorrect results. (This kind of incompatibility is not uncommon between languages.) On the other hand, if the Ada implementation is designed to support interaction with FORTRAN software, it may store arrays in the manner required by FORTRAN, or it may give a means to specify how data are represented so no problem will arise.

[23]While the Ada language provides special features (representation clauses and implementation-dependent pragmas) that allow software developers to specify data storage layouts in detail, at present not all Ada compilers adequately support these facilities. See Section 4.2.2.4.

To summarize, here are some questions to ask about run-time support:

- How much run-time support code is associated with the use of particular Ada language features?
- Is run-time support software loaded whether or not it is actually needed?
- Is some of the run-time support essentially the same for both Ada and the other language? If an overlap is significant, is it necessary to use the associated Ada features?
- Can the run-time environments co-exist? Various languages assume different system memory models or have different paradigms regarding the scheduling of tasks. This type of incompatibility may cause unpredictable system behavior.

7.2. Isolating Subsystems

To minimize language incompatibility, the portion of the system that is written in one language should be relatively independent of the portion written in another language.

It may be appropriate to write a new portion of a system in Ada if the new portion is relatively isolated from the unmodified portion. Two examples where Ada might be successfully introduced are:

- **DBMS/file system interfaces**: Suppose the unmodified portion of a system is a database management system (DBMS) written in COBOL or purchased from a vendor (so its source code is not available). If the remainder of the system consists of software that accesses the DBMS and processes the results of DBMS queries, then calls to the DBMS can probably be written as easily in Ada as in any other language.
- **Distributed systems**: On a distributed system, it should be possible to replace the code which exists on individual processor node(s) as long as the system-wide interfaces are maintained.

7.3. Replacing the Whole System

Indicators that perhaps a system should be totally reimplemented in Ada include continual trouble reports, significant software change, or new target hardware.

If a system is operating smoothly, replacing that system just for the sake of having it in Ada is generally *not* recommended. However, there are several rules of thumb which may indicate the need for a system replacement. Assuming the availability of tools for the target processor, Ada should be the replacement language if any of the following situations exist:

- **Poor overall system reliability**: Fixing problems creates more problems, or unresolvable trouble reports exist. An additional factor might be that the system has had cumulative changes amounting to 65% from a significant baseline. This would indicate a fragile structure which would become increasingly frail with each successive set of changes.
- **35% one-time change**: More than 35% of the system has to be changed at any one time, and the changes are spread throughout the system.
- **Hardware replacement**: Hardware upgrade or replacement provides an opportunity to also upgrade the software design to take advantage of new hardware capabilities. In general, old software running on new hardware will not take advantage of improved hardware capabilities such as additional memory. As such, maximum performance improvement will not be achieved.

7.4. Translating Languages

Automatic translation from one programming language to another, while a promising idea, should be approached cautiously.

Automatic translation (through the use of computer-based language translation tools) of existing code into Ada is sometimes proposed when an existing system is upgraded. While a number of translation systems exist, and translation offers the possibility of getting something running quickly, there are significant problems and issues to be considered before taking this approach:

- **Degree of translation**: There are always some inherent incompatibilities between any two languages.[24] Not all constructs in other languages can be translated automatically to Ada with complete accuracy. A good translation system will flag possible mistranslations so they can be manually inspected. If the existing software contains many problematic constructs, much manual effort will be required to detect and fix possible mistranslations.
- **Loss of software engineering benefits**: The translated code will most likely be harder to maintain than the original software and certainly harder to maintain than a system designed and written in Ada from the start. An example problem area is readability — automatic translators may generate strange names for variables. In addition, Ada allows a software developer to express intent more fully than other languages. For example, Ada allows subroutines and data to be organized into groups called *packages*, which improve software understandability and modifiability. If such a grouping is not present in the original system, an automatic translation system cannot provide it.
- **Special routines**: Depending on the language being translated, there may be special functions and capabilities such as those provided by the run-time system that are not directly translatable. These special functions and capabilities may require that significant new code be written to mimic them in the new language, at unknown consequences in program schedule and performance of the resulting software.
- **Testing**: Translation does not guarantee that the resulting code will function in a manner comparable to the original code. For example, comparable language constructs may *not* be implemented with comparable efficiency. In addition, the translated code may stress the compiler in unusual ways, revealing new compiler bugs. In short, rigorous functional and performance testing are required on the translated code. If acceptance tests exist for the original system, they should be used.

In summary, language translation must be viewed as a risky, short-term proposition which will yield results of unknown quality. Redesigning and recoding the system in Ada or interfacing newly written Ada code with the current implementation are much preferred approaches.

If a translation strategy is being considered, an in-depth investigation of commercial and public domain translators is strongly suggested as part of the feasibility study. This investigation should use a robust set of test cases and benchmarks which are typical of the particular application.

---

[24] Some languages, notably FORTRAN, have many different dialects, which makes translating even between dialects of the same language difficult.

8. Software Reuse and Ada

This chapter examines:

- why reuse is of interest
- a brief history of reuse
- how Ada can help with reuse
- a plan for making reuse a reality
- management and contractual considerations.

8.1. Why Is Reuse of Interest?

> *Many subfunctions in new software systems are similar, if not identical, to those in previously developed systems. If software were properly designed, those subfunctions could be reused to produce new systems faster, more reliably, and at lower cost.*

Writing the same or similar code over and over is time consuming and costly. For example, Army C3I systems could potentially save time and money by reusing the graphics display software that has been rewritten for many systems [8]. In addition to improved productivity and reduced cost, software reuse:

- increases reliability because reused code and its accompanying design will have been extensively exercised and tested, and
- can facilitate rapid prototyping [35].

8.2. Brief History of Reuse

> *Software reuse is not a new concept; but until now, impediments prevented it from becoming a reality.*

Reuse has been practiced before with other languages on an informal, ad hoc basis. There are a number of reasons reuse was not practiced formally in the past:

- **Absence of incentives**: Technically, systems were not sophisticated or large enough to warrant reuse, and reliability requirements were generally lower than they are today. In addition, there were few, if any, procurement incentives [35].
- **Immaturity of software development process**: Software was regarded as an unmanageable craft.
- **Technically limited languages**: Early languages had few features which facilitated the development of reusable components.
- **Lack of language standardization**: FORTRAN, for example, with its many dialects, varied (in some cases significantly) across computer systems and compilers. It was probably more convenient, if not less risky, to rewrite rather than reuse code.
- **Cultural and societal impediments**: Our educational system encouraged reinventing rather than reusing [35].
- **Problems with indexing and retrieval**: Lack of information-retrieval capabilities make it difficult to locate needed pieces of code.

- **Efficiency**: Due to historical concerns about resource-limited computers, custom-tailored code was the prevailing approach. Only in recent years, as processors have advanced significantly in memory availability and processing power, has performance been traded for productivity and reliability.

8.3. How Does Ada Support Reuse?

Ada's standardization and some of its special features mean that Ada is especially suitable for promoting software reuse. However, Ada does not automatically make software reusable. Rather, facilities are available to make reuse possible.

Ada has helped overcome some of the historical barriers to reuse. Ada's language standardization and unique facilities such as packages, generics, and separate compilation have provided a catalyst for reuse. There are documented cases where reuse helped bring projects in on time and/or ahead of budget [6, 7]. These hold the promise for the future.

Ada allows interfaces to be precisely defined so functions performed by a software unit are obvious, precise, and regulated. These capabilities, if properly used, will facilitate the reuse of software components. However, Ada does not *automatically* make software reusable; rather, facilities are available to design a component for reuse. Indeed, programming for reuse requires a higher level of effort than standard programming.

8.4. Reuse: Action Plan

In the short term, several minimal-cost actions can be implemented within the scope of an individual program:

- **Code sharing**: Engineers should be encouraged to share code (among those assigned to a project and among different projects) and thus reduce workload.
- **Commercially available software packages**: Software for the manipulation of various data structures can be purchased.
- **Methodology**: Reuse can be made an explicit program objective. For example, potentially reusable components could be identified as part of the design process and then examined and reviewed as part of software review activities. This approach would not require much additional work if the information already existed as the result of a well-applied design process.

In the long term, several actions are possible to take advantage of the collective efforts and experience of several projects. The following ideas will potentially yield dramatic productivity improvements. These improvements may be beyond the scope of an individual project and thus will require either additional government funding or research and development activities within various organizations:

- **Initial automated libraries**: A library of reusable components (using the components identified as part of the previous action items) can be developed and maintained. Extra attention should be placed on quality documentation for the library, and a means of modification and user feedback should be provided. In addition, technical concerns regarding memory usage, run-time performance, and software engineering quality should be addressed on a component basis.
- **Fully integrated libraries**: A fully integrated library system could be developed to categorize components into a "software parts catalog" based on various attributes. Software

engineers could automatically retrieve components based on specifications, much the way electrical engineers access and retrieve components from hardware data books.

- **Composition systems**: Composition systems would possess knowledge of an application domain and not only retrieve, but also set parameters, tailor, and interconnect several related pieces of code. Some pioneering work in this area is being accomplished by the Common Ada Missile Packages (CAMP) effort, sponsored by Eglin Air Force Base, Florida.

8.5. Potential Considerations

Contractual and management considerations are very important to the future of reuse.

Before extensive reuse can become reality, contractual and management considerations must be addressed, including:

- **Data rights**: Who owns the original software, especially that which was produced under contract?
- **Cost estimating**: What impact will reuse have on current software cost-estimating procedures, which typically are based on the number of original lines of code produced?
- **Bug fixes**: Who fixes bugs in reused code?
- **Contract liability**: If a contractor has problems making reused code work, where does the responsibility fall?
- **Incentives**: Who pays for the effort (including additional testing) to make software reusable? What cultural changes (both inside institutions and within defense acquisition procedures) will be necessary to develop incentives in this area?

9. Learning Ada: Training Implications

9.1. Rationale for Ada Training

Maximizing the benefits of Ada requires knowing the software engineering techniques Ada supports as well as the language. Because it is important to teach software engineering along with the language, training costs associated with the introduction of Ada will be higher than for other languages. Of course, Ada training will vary, as not everyone needs the same level of expertise in Ada and software engineering.

Appropriate training is required to realize Ada's positive impact on schedule, cost, and quality goals.[25] An effective training program should be based on the following points:

1. Using Ada effectively requires more than just knowing the language — it requires understanding the software engineering techniques the language supports. For example, data abstraction and information hiding, while easy to understand in principle, may present some difficulty in application [21]. As a result, quality Ada training must be different from and more extensive than training for other languages. While Ada training will cost more initially because more will be taught, in the long run, the use of Ada and its associated software engineering methods can reduce individual program development and maintenance costs.
2. Software designers, system engineers, Ada programmers, and software managers in both industry and government *all* need to know something about Ada and its associated software engineering techniques. Ada training should address all these groups, but the depth and kind of knowledge needed will be different for each. But because of Ada's new concepts, the learning curve for all concerned can be steep.
3. Management training is especially critical. To capably manage and direct programs involving new technology, program managers must understand how to apply the technology [21] and how it differs from existing approaches.

9.2. Audiences and Course Content

All personnel need some training in Ada and its associated software engineering methods to obtain the maximum benefits from using Ada. In addition, an Ada training program should always consider the background of students.

Following are some guidelines about target audiences and key content material:

- The background of students attending the classes must be considered in course design. For example, software engineers who know only languages such as FORTRAN or assembler must have extensive training in Ada since they will have many new concepts to learn. Such programmers can be easily overloaded in training courses that are too intensive and too short. Initial training for these programmers can take 3 weeks; becoming truly productive using Ada can take 4 to 6 months of experience. Those who already know a structured high-order language like PASCAL should assimilate the new techniques more rapidly.
- Engineers concerned with software design will need training in the use of Ada-based program design languages (PDL).

---

[25] The Navy's Design Evaluation Tool Program (see Appendix II) experienced a 12-month schedule slip because of inadequate training [32].

- Software engineers developing Ada software for embedded systems need to understand how the compiler and tools in use on a project affect memory and throughput constraints.
- System engineers responsible for hardware and software integration will need a deep understanding of the systems engineering impact of HOL selection [27].
- Software testing personnel need to understand how new Ada language constructs such as packages and tasks affect testing approaches.
- Managers should have an understanding of software engineering principles, Ada concepts, hardware and software, the impact of methodologies on the software architecture, the impact of Ada on the software and system lifecycle, and the costs and capabilities of tools.
- Personnel attending design reviews should have some understanding of how Ada will affect the software architecture.
- Training requirements should also be levied by the primary contractor upon all subcontractors. This will help ensure continuity in methodology, standards, and approach across the entire development effort [21].

9.3. Guidelines for Evaluating Training Programs

The most important aspects of any Ada training program are a software engineering emphasis and hands-on practice.

Following are some guidelines for evaluating Ada training programs:

- **Tailoring**: Training can be purchased, or developed internally. Curricula developed internally or customized for an organization tend to be more effective because they:
  - provide for curriculum coordination
  - can embody specific methodologies and standards
  - can be oriented toward specific tools and hardware considerations
  - are generally more responsive to sudden, short-term needs.
- **Emphasis**: The emphasis of courses should be on understanding what the language can do and how it should be used to produce well-engineered systems. Courses which only teach the syntax and semantics (the grammar) of Ada should be avoided.
- **Length and topics**: For any introductory course, a *minimum* of 5 class days is required to teach the basics of the language and to introduce the underlying software engineering principles. Additional classes, which include Ada's more sophisticated features and implementation tradeoffs, will be required for people who will use Ada heavily, who do not have previous HOL background, or who will be responsible for design activities. Refresher courses should be offered periodically after the initial training.
- **Equipment**: All training (including refresher courses) should use validated Ada compilers. Although using nonvalidated or subset compilers may appear convenient or cost effective in the short term, productivity on the job will almost certainly suffer when students have to familiarize themselves with the full language. (To some degree, due to code work-arounds, the same thing is true if very immature validated compilers are used for training.)
- **Documentation**: In addition to handouts of course materials, model solutions to problems should be available; they are an especially useful way to teach the Ada language and its underlying principles.

- **Hands-on lab**: It is *imperative* that hands-on programming exercises be a major component of the training process. Just as it is difficult to learn how to repair radar sets or jet engines without practical, applied work, the same is true for building computer software and learning computer languages. In fact, the more detailed the training, the more imperative hands-on exercises become. For example, in classes dealing with using Ada for a specific embedded processor, there is no substitute for writing software with the compiler and tool set that will be used on the actual development program. The software engineers working at this level need to understand the performance and limitations of a particular tool set.
- **Monitoring progress**: The training program should have a method for measuring student progress and providing feedback to students.
- **Instructor availability**: A trained instructor must be available to help students and provide guidance when required.
- **Computer-assisted instruction (CAI) and video tapes**: These are good *supplemental* tools for teaching language syntax and providing overviews and review. However, they should *not* be the focus of the training effort because they generally do not have a software engineering emphasis.
- **Real project use:** To achieve maximum training affect, personnel should be assigned immediately to an Ada project where experienced Ada software engineer(s) are available for consultation for the first 4 to 6 months.

9.4. Getting More Information

Training courses are available for a variety of audiences. More information is available from:

- **Ada Information Clearinghouse (AIC)**: AIC has information about upcoming classes, seminars, etc. For more detailed information, the AIC publishes the *CREASE: Catalog of Resources for Education in Ada and Software Engineering*. Refer to Appendix I for contact information.
- **Ada-JOVIAL Newsletter**: The newsletter of the Ada-JOVIAL users group (AdaJUG) is published by the Language Control Facility at Wright Patterson AFB, Ohio. Several pages in each issue are allocated to product and training announcements.

Appendix I
Ada Working Groups and Agencies

Many groups are involved in various Ada technology activities. In the following sections, details of various groups are discussed, including:

- professional organizations
- U.S. government sponsored/endorsed organizations
- Non-U.S. government organizations
- service program managers

I.1. Professional Organizations

AdaJUG: The Ada-JOVIAL users group is a national organization composed of defense contractors and government representatives. The Ada-JOVIAL Newsletter is published by:

ASD/SIOL
Computer Operations Division
Information Systems and Technology Center
Wright Patterson AFB, Ohio 45433-6503
(513) 255-4472/4473

SIGAda: The Association for Computing Machinery Special Interest Group on Ada is a professional association composed of people interested in the Ada language. Ada Letters is the SIGAda newsletter. The following SIGAda groups are examining issues of particular interest:

ARTEWG: Ada Run-Time Environment Working Group

- **Purpose**: Investigates Ada run-time issues; establishes conventions, criteria, and guidelines; develops strategies for improving run-time systems so that they can be tuned to individual applications and projects.
- **Contact**:

  Mike Kamrad
  Honeywell Systems and Research Center
  Minneapolis, Minnesota
  (612) 782-7321

PIWG: Performance Issues Working Group

- **Purpose**: Investigates Ada compiler performance issues, devises benchmark tests in areas such as exception handling, loop overhead, procedure calls, task creation, task rendezvous; and collects and disseminates results.
- **Contact**:

  Jon Squires
  Westinghouse Electric Corporation
  Baltimore, Maryland
  (301) 765-3748

SDSAWG: Software Development Standards and Ada Working Group

- **Purpose**: Investigates the interaction of Ada with DoD-STD-2167.
- **Contact**:

Don Firesmith
Magnavox Electronic Systems Corporation
Ft. Wayne, Indiana
(219) 429-4327

I.2. U.S. Government Sponsored/Endorsed Organizations

AJPO: Ada Joint Program Office

- **Purpose**: Oversees the total direction of the Ada program.
- **Contact**:

Virginia Castor, Director
Ada Joint Program Office
the Pentagon, Room 3E114
Washington, D.C. 20301-3081
(202) 694-0210

Ada Board:

- **Purpose**: A federal advisory committee, composed of compiler developers, language designers, embedded system users, and government personnel chartered to advise the AJPO Director.
- **Contact**:

Virginia Castor, Director
Ada Joint Program Office
the Pentagon, Room 3E114
Washington, D.C. 20301-3081
(202) 694-0210

Ada Information Clearinghouse (AIC):

- **Purpose**: Supports the AJPO via the distribution of Ada-related information, including lists of validated compilers, classes, conferences, text books, and programs using Ada An electronic bulletin board is available at (202) 694-0215. A newsletter is also available.
- **Contact**:

3D139 (1211 S. Fern, Rm. C-107)
the Pentagon
Washington, D.C. 20301-3081
(703) 685-1477 or (301) 731-8894

Ada Compiler Validation Capability (ACVC):

- **Contact**:

Georgeanne Chitwood
JLCF/ASD/ADOL
Wright-Patterson AFB, Ohio 45433
(513) 255-3813

Ada Language System/Navy (ALS/N):

- **Purpose**: Directs the development of the Navy's Ada environment effort which will be producing compilers for the AN/UYK-43, AN/UYK-44 and AN/AYK-14 computers.
- **Contact**:

  Bill Wilder
  Naval Sea Systems Command (PMS-408)
  Washington, D.C.
  (202) 692-8204

ASEET: Ada Software Engineering Education and Training Team

- **Purpose**: Examines issues in providing quality and timely Ada education and training.
- **Contact**:

  Major Douglas Samuels
  Keesler AFB
  Biloxi, Mississippi

AVO: Ada Validation Organization

- **Purpose**: Implements compiler validation policy, overviews development of the Ada Compiler Validation Capability (ACVC).
- **Contact**:

  Audrey Hook
  Institute for Defense Analyses
  1801 Beauregard Street
  Alexandria, Virginia 22311
  (703) 824-5501

E&V: Evaluation and Validation Team

- **Purpose**: The Ada community needs the capability to assess APSEs (Ada Programming Support Environments) and their components and to determine their conformance to applicable standards (e.g., DOD-STD-1838, the CAIS Standard). The technology required to fully satisfy this need is extensive and largely unavailable; it cannot be acquired by a single government-sponsored, professional society-sponsored, or private effort. The APSE Evaluation and Validation (E&V) task provides a focal point for addressing these needs by:

  1. identifying and defining specific technology requirements
  2. developing selected elements of the required technology
  3. encouraging others to develop some elements, and
  4. collecting and distributing information to DoD components, other government agencies, industry, and academia [33].

- **Contact**:

  Raymond Szymanski
  AFWAL/AAAF
  Wright-Patterson AFB, Ohio 45433
  (513) 255-2446

KIT: KAPSE Interface Team

- **Purpose**: Investigates issues in the development of Ada program support environments, especially standardization strategies for interfaces that will promote portability of software and data. Responsible for the development of the CAIS (Common APSE Interface Set).
- **Contact**:

  Patricia Oberndorf
  Naval Ocean Systems Center
  421 Catalina Blvd.
  San Diego, California 92152
  (619) 225-6682

PDL activities:

- **Purpose**: Conducts Ada PDL research.
- **Contact**:

  Larry Lindley
  Naval Avionics Center
  Indianapolis, Indiana

I.3. Non-U.S. Government Organizations

Ada-Europe:

- **Contact**:

  Knut Ripken
  (Paris) 33-1-47717232

Ada-Sweden:

- **Contact**:

  Elsa-Karin Boestad-Nilsson
  (Stockholm) 46-8-631500

I.4. Service Program Managers

The service program managers are responsible for Ada policy, activities, and implementation plans within each service.

U.S. Air Force:

- **Contact**:

  Col. Casper H. Klucas, Program Manager
  HQ AFSC/PLR
  Andrews AFB, Maryland 20331
  (301) 981-5731

U.S. Army:

- **Contact:**

  Lt. Col. David R. Taylor, Program Manager
  AMCDE-SB
  5001 Eisenhower Avenue
  Alexandria, Virginia 22333
  (703) 274-9309

U.S. Navy:

- **Contact:**

  Lt. Cmdr. Philip Myers, Acting Program Manager
  Space & Naval Warfare Systems Command
  SPAWAR 3214
  Washington, D.C. 20363-5100
  (202) 692-9208

Appendix II
Programs Using Ada

The following information about programs using Ada has been selected and adapted from several sources [2, 3, 32] should not be considered a complete list. For further information, contact the Ada Information Clearinghouse as indicated in Appendix I. Each of the following entries includes:

- program name
- system status:
  - PLN: being planned
  - DEV: under development
  - CMP: completed or operational
- number of lines of Ada code
- kind of software being written:
  - CC: command/control
  - EMB: embedded
  - SIM: simulation programming
  - SUP: support software
- target computer (where known)
- development organization (where known)
- brief abstract.

The entries are organized according to the kind of software being written.

II.1. Army Programs

Advanced Field Artillery Tactical Data System (AFATDS) DEV 770K CC MC68020
(Magnavox)

Abstract: Automated command and control system intended to serve as both a subordinate system and objective control element of the Fire Support functional system. AFATDS will replace TACFIRE [17].

Maneuver Control System (MCS) CMP 34K CC —
(Ford Aerospace)

Abstract: A command and control program employed in Army training exercises.

Regency NET DEV 65K EMB —
(Fort Monmouth)

Abstract: Provides automated functions on a distributed basis for the RN HF radio communication system.

FLIR Mission Payload Subsystem (FMPS) DEV 11K EMB —
(MICOM)

Abstract: Controls such functions as auto-tracking a target, ranging or designating with the laser, and responding to manual sight-line control commands from the ground.

Light Helicopter Experimental (LHX) PLN — EMB —
(AVSCOM, St Louis — Glen Tomlin, (314) 263-1813)

Abstract: Integrated avionics systems for the Army's next-generation helicopter.

Mobile Automated Field Instrumentation System (MAFIS) DEV 64K SIM —
(Fort Hood)

Abstract: Real-time monitoring of simulated weapons engagements. Only the command and control subsystem of MAFIS is being written in Ada.

Intermediate Forward Test Equipment (IFTE) DEV 400K SUP —
(Fort Monmouth)

Abstract: System software will include the run-time system software, an ATLAS compiler, simulation software, and maintenance/self-test/diagnostic software.

Army Test Program Set Support Environment (ATSE) DEV 200K SUP —
(Fort Monmouth)

Abstract: An interactive, automated environment for Test Program Set development and management.

Robotized Wire Harness Assembly System (RWHAS) CMP — SUP —
(Redstone Arsenal)

Abstract: Program produces data used by robots to manufacture a wire harness.

II.2. Navy Programs

F4-J Weapon System Trainer CMP 150K EMB Gould 32/8780
(Science Applications International Corporation)

Abstract: F4-J real-time trainer software was converted from FORTRAN to Ada. The final Ada version requires less memory space than the FORTRAN version. Naval test personnel were unable to distinguish between the flight performance and operation of the Ada and FORTRAN versions.

Design Evaluation Tool DEV 100K SIM —

Abstract: A modifiable simulator of the AN/UYK-43 computer.

Ada Language System/Navy DEV 250K SUP —

Abstract: Provides program generation and execution support for mission-critical software targeted to Navy standard embedded computers.

NOSC Tools CMP 150K SUP —

Abstract: A set of APSE tools, including an Ada-based PDL, pretty printer, data dictionary, standards checker, configuration management system, symbolic debugger, etc.

APSE Interactive Monitor(AIM) CMP 25K SUP —
(Texas Instruments)

Abstract: Research investigation into program support environments, interfaces, software portability, and interoperability [6].

Aircraft Wire Harness Manufacturing System DEV 3K SUP —

Abstract: Data generator, manufacturing data manager, and real-time communications with an engineer's workstation and file server for a semi-automatic wire harness assembly center.

II.3. Air Force Programs

MILSTAR DEV 500K CC —
(Lockheed)

Abstract: A joint military communications system for worldwide strategic and tactical use.

Global Decision Support System (GDSS) DEV 45K CC —
(Scott AFB)

Abstract: Wide-area network replicated database system for command and control of military airlift command resources.

Air Force Support to MEECN DEV 12K EMB Intel 8086
(Offutt AFB)

Abstract: Software controls a transmitter that sends messages from the National Command Authority to strategic defense forces.

E-4B Message Processing System PLN 20K EMB ROLM Hawk/32
(Tinker AFB)

Abstract: Message processing software.

OFP for F-16 DE/CIS CMP — EMB —
(General Dynamics)

Abstract: An operational flight program (OFP) for the F-16 Data Entry/Cockpit Interface System (DE/CIS) was written in Ada.

F-20 Operational Flight Program CMP — EMB 1750A
(Northrop Aircraft)

Abstract: Existing JOVIAL code for the F-20 MC/OFP was translated to Ada. Real-time functions include Pylon Interface, Communications Navigation Identification Interface Unit (CNIIU), display system, radar, Integrated Navigation System (INS), and missile interface.

HOL Electronic Warfare Software Analysis DEV 5K EMB —
(Wright-Patterson AFB)

Abstract: Rewrite of an electronic warfare subsystem in Ada.

Tactical Ada Guidance (TAG) CMP 3K EMB —
(Eglin AFB)

Abstract: Investigates Ada's applicability to tactical missiles, including a comparison with existing JOVIAL code.

Autonomous Synthetic Aperture Radar Guidance (ASARG) DEV — EMB —
(Eglin AFB)

Abstract: Processes digitized radar image in real time, controls SAR updates to missile navigation system, and controls antenna steering.

Real-Time Target Identification (RTID) DEV — EMB —
(Eglin AFB)

Abstract: Exploratory development of a target state estimator. Estimator algorithm will be downloaded to a microprocessor interfaced with 6 degrees-of-freedom missile simulation.

Small ICBM DEV 2K EMB 1750A
(Norton AFB)

Abstract: Real-time flight communications and telemetry processing.

Adv. Processor Technology for Air-to-Air Missiles DEV — EMB 1750A
(Eglin AFB)

Abstract: APTAAM is software for command and control of a missile system including multimode radar processing, guidance, and head control. The software controls four data processors and interfaces to an advanced signal processor.

Ada Target Sensor Subsystem (ATSS) Study DEV — SIM —
(Wright-Patterson AFB)

Abstract: Development of a radar model in Ada and JOVIAL.

TAC Weapons System Evaluation Program (TAC WSEP) DEV 35K SIM —
(Tyndall AFB)

Abstract: Graphic representation of data from the Weapons System Evaluation Program Database, plus simulations based on this data. Additional software for reducing real-time data from target drones and air-to-air missiles.

Ada SAM Missile Simulation CMP 40K SIM —
(Wright-Patterson AFB)

Abstract: SAM missile simulation, which models a particular threat SAM, was validated by comparison with a FORTRAN simulation of the same missile.

Mobile Information Management System (MIMS) DEV 15K SUP —
(Strategic Air Command)

Abstract: A relational information management system with a fourth-generation application and query language.

Tester Independent Support Software System (TISSS) DEV 80K SUP —

Abstract: Automates generation of microelectronic MIL-M-38510 and product specifications. Automatically generates test programs. Maintains all design and test data in a central database.

Standard Automated Remote to AUTODIN Host (SARAH) DEV 20K SUP —
(Gunter AFB)

Abstract: Provides JANAP 128 and DD173 message preparation, storage on floppy disk, and transmission/reception through AUTODIN.

Guidance Instruction Set Architecture (GISA) DEV 6K SUP —
(Eglin AFB, FL)

Abstract: Assembler and functional simulator for an experimental microprocessor instruction set architecture.

II.4. Commercial and IRAD Programs

Secondary Flight Control Computer DEV 7K EMB —
(The Boeing Company)

Abstract: Secondary flight control for the 7J7 aircraft.

Digital Stall Warning Computer CMP 2K EMB —
(The Boeing Company)

Abstract: Demonstrator program that reimplements digital stall warning software using Ada. The DSWC system is used for the 737-300 aircraft.

Beech Starship Avionics Program CMP 82K EMB Intel 80186
(Rockwell Collins)

Abstract: Flight management system and electronic flight display developed for the Beech Starship Aircraft. Rockwell Collins felt that the additional time required to perform compilations using Ada was more than compensated for by the resulting reduction in debugging time [3].

Senso Station Simulation Facility DEV 25K SIM —
(Allied Signal Corporation)

Abstract: Simulation of signal processing and display for a helicopter senso station.

FAST — Flexible Ada Simulation Tool CMP 20K SIM —
(Ford Aerospace and Communications Corporation)

Abstract: General-purpose, discrete-event simulation capability enhanced with interactive environment and concurrent display of simulation output statistics.

Shell Oil Geophysical Package DEV 30K SUP —
(Shell Oil)

Abstract: A series of complex primitives used in geophysical engineering.

Fault-Tolerant Distributed Operating System DEV 22K SUP 1750A
(Westinghouse Electric Corporation)

Abstract: The distributed operating system functions include process scheduling, communications, input handling, and fault tolerance for a distributed system consisting of three 1750A processor modules communicating over a 16 bit transaction bus.

RELATE/3000/ProjectALERT CMP 200K SUP —
(CRI, Inc.)

Abstract: RELATE/3000 is a relational database management system written in Ada. ProjectALERT is an automated project management system based on CPM and GANT project management methodologies.

Rational Development Environment CMP 1M SUP —
(Rational)

Abstract: The Rational Environment is a universal host software development, hosted on the Rational R1000.

Application Generator for Space Vehicle Control Systems DEV 26K SUP —
(Boeing)

Abstract: Prototype application generator providing a high-level graphical approach to software generation.

Ada Manager/Ada Quest Version 1 CMP 15K SUP —
(AdaSoft, Inc.)

Abstract: A single-user relational DBMS intended for use in Ada workstation environments.

AutoCode/Ada CMP 25K SUP —
(Integrated Systems, Inc.)

Abstract: Automatic Ada code generator for real-time software. The system allows generation of Ada code for multi-rate periodic and asynchronous real-time control systems.

Appendix III
Ada Textbooks

For those interested in learning more about the particulars of the Ada language, the following list is provided. No endorsement of any text is implied.

Ada for Experienced Programmers
A. Nico Habermann, Dewayne E. Perry
Addison-Wesley; 1983; $20.95

Ada for Programmers
Eric W. Olsen, Stephen B. Whitehall
Reston; 1983; 310 pages; $20.95

Ada: A Programmer's Conversion Course
Michael J. Stratford-Collins
John Wiley & Sons; 1982; 192 pages; $48.95

Ada as a Second Language
Norman Cohen
McGraw Hill; 1985

Ada: An Introduction
Sabina H. Saib
Holt, Rinehart, and Winston; 1985; 350 pages; $25.95

Ada: Concurrent Programming
Narain Gehani
Prentice-Hall; 1984; 272 pages; $24.95

Ada: An Advanced Introduction Including the LRM
Narain Gehani
Prentice-Hall; 1984; 672 pages; $29.95

An Introduction to Ada
Stephen Young
John Wiley & Sons; 1983; 320 pages; $29.95

Beginning Programming with Ada
James A. Saxon, Robert E. Fritz
Prentice-Hall; 1983; 240 pages; $16.95

Concurrent Programming in Ada
A. Burns
Cambridge University Press; 1985; $29.50

Developing Large Software Systems Using Ada
Sommerville/Morrison
Addison-Wesley; 1985; 400 pages; $23.95

Introduction to Ada
David Price
Prentice-Hall; 1984; $22.95

Parallel Programming in ANSI Standard Ada
George W. Cherry
Prentice-Hall; 1984; $21.95

Problem Solving with Ada
Brian Mayoh
John Wiley & Sons; 1982; 240 pages; $17.90

Programming Embedded Systems with Ada
V.A. Downes, S.J. Goldsack
Prentice-Hall; 1982; 400 pages; $30.00

Programming in Ada (2nd Edition)
J.G.P. Barnes
Addison-Wesley; 1983; $19.95

Programming in Ada
Richard Wiener, Richard Sincovec
John Wiley & Sons; 1983; 345 pages; $25.95

Programming in Ada: A First Course
Robert G. Clark
Cambridge University Press; 1985; $17.95 paperback; $39.50 hardcover

Software Engineering with Modula-2 and Ada
Richard Wiener, Richard Sincovec
John Wiley & Sons; 1984; 480 pages; $24.90

Software Components with Ada
Grady Booch
Benjamin/Cummings; 1987; $35.95

Software Engineering with Ada (2nd Edition)
Grady Booch
Benjamin/Cummings Company; 1987

Studies in Ada Style (2nd Edition)
P. Hibbard et al.
Springer-Verlag; 1983

Systems Design with Ada
R.J.A. Buhr
Prentice-Hall; 1984; $21.95

The Ada Programming Language
Sabina H. Saib, Robert E. Fritz
IEEE Computer Society Press; 1983; $30.00

The Ada Programming Language (2nd Edition)
I.C. Pyle
Prentice-Hall; 1985; 336 pages; $23.95

Understanding Ada
Ken Shumate
Harper & Row; 1984; $18.95

Appendix IV
Ada Compilers for Target Processors

The following information was taken from the AJPO validated Ada compilers list dated 1 May 1987, and is limited to compilers validated for use on various embedded processors. Other larger machines and PC products have been omitted from this subset list. If a processor is not included in this list, there were no validated compilers for it at the time of publication of this handbook.

PROCESSOR	VENDOR	HOST MACHINE & OS	TARGET MACHINE & OS
1750	Advanced Computer Techniques Corp. (New York)	VAX-11/785 (under VMS 4.2)	Fairchild 9450/1750A (bare machine)
	Intermetrics,Inc. (Boston)	VAX-11/785 (under VMS 4.2)	ECSPO SIM50A Release R0304-4.000 — a MIL-STD 1750A simulator (no operating system)
	Systems Designers	DEC VAX 8600 (under VMS 4.2)	MIL-STD-1750A implemented on the Ferranti 1750A Computer System 100A (bare machine)
	Verdix Corp.	MicroVAX II (under VMS 4.2)	Fairchild 9450 (Bare machine)
1666B	ROLM Mil-Spec Computers	Data General Eclipse MV/10000 (under AOS/VS, Rev. 6.03)	ROLM 1666B, 1666C (under RMX/RDOS, Rev 3.42)
Hawk/32	ROLM Mil-Spec Computers	Data General MV/8000 (under AOS/VS, Rev. 6.03)	ROLM Hawk/32 (under ARTS/32, Rev. 2.07 and AOS/VS, Rev.6.03)
8086 80186	SofTech, Inc.	VAX-11/780 and VAX-11/785 (under VMS 4.1)	INTEL 8086 on 86/30 board, and INTEL 80186 on 186/03A board

PROCESSOR	VENDOR	HOST MACHINE & OS	TARGET MACHINE & OS
80286	CAP Industry Ltd. (England)	DEC VAX 8800 (under VMS 4.4)	Intel 80286 (bare machine)
68000 family	ALSYS	VAX 11/750 (VMS 4.2)	Motorola 68020 (under VRTX)
	Systems Designers	DEC VAX 8600 (under VMS 4.2)	MC68010, implemented on MVME 117-3FP board (bare machine); Motorola MC68020 (bare machine)
	TeleSoft, Inc.	MicroVAX II (under Micro VMS 4.2)	Motorola 68020, Motorola 68010, & Tektronix 8540 (with M68010 CPU) (all bare machines)
	Verdix Corp.	DEC VAX 11/750 (UNIX 4.2 BSD) DEC MicroVAX II (MicroVMS 4.2)	Microbar GPC68K (bare machine)
MicroVAX/MilVAX	SofTech, Inc. AdaVAX	VAX 8600, VAX-11/785, & 780 (VMS 4.1) MicroVAX II (MicroVMS 4.1M)	All Host Configurations
	Digital Equipment Corp.	VAX 8800, 8700, 8650, 8600, 8500, 8300 & 8200, VAX 11/785, 782, 780, 750 & 730 (VAX/VMS V4.4) MicroVAX II and VAXstation II (MicroVMS V4.4)	All Host Configurations and MicroVAX II (under VAXELN Toolkit, V2.2, in combination with VAXELN Ada V1.1)
	TeleSoft, Inc.	MicroVAX II Model 630-QY (MicroVMS 4.1) VAX-11/780 (under VMS 4.1)	Same as Host

References

[1] Ada Goes to Work. *Information Week,* September 8, 1986, pp. 35-44.

[2] Ada Information Clearinghouse. *Newsletter.* August 1986.

[3] Ada Information Clearinghouse. *Newsletter.* December 1986.

[4] Ada Joint Program Office. Ada Compiler Validation Procedures and Guidelines. *Ada Letters,* March/April 1987, pp. 28-57.

[5] Bryce Bardin. Personal communication. April 1987.

[6] Jerry Baskette. Life Cycle Analysis of an Ada Project. *IEEE Software,* January 1987, pp. 40-47.

[7] G. Booch. *Ada Systems for Reuseable Components.* February 1987. IEEE COMPCON, San Francisco.

[8] Harold C. Brooks. Personal communication. January 1987.

[9] Virginia Castor. *Guidelines for the Evaluation of Technical Proposals from the Ada Perspective (Draft).* April 1985. AFWAL/AAAF, Wright Patterson Air Force Base, Ohio.

[10] Terry Courtwright. Personal communication. August 1985.

[11] *DoD Directive 3405.1, Computer Programming Language Policy.* April 2, 1987.

[12] *DoD Directive 3405.2, Use of Ada in Weapon Systems.* March 30, 1987.

[13] Donald G. Firesmith. Should the DoD Mandate a Standard Software Development Process? *Proceedings of Joint Ada Conference 1987,* pp. 159-167. March 1987. Also published in Defense Science and Electronics, Vol 6 Number 4, April 1987, pp. 60-64.

[14] John Foreman. *Building Software Tools in Ada - Design, Reuse, Productivity, Portability.* July 1985. Presented at AdaJUG, Baltimore.

[15] J. Kaye Grau, Kathleen A. Gilroy. Compliant Mappings of Ada Programs to the DoD-STD-2167 Static Structure. *Ada Letters,* March/April 1987, pp. 73-84.

[16] Charles Hammons. What Is a Million Lines of Code? *Ada Rendezvous,* Spring/Summer 1986, pp. 23-24. Published by Texas Instruments, Military Computer Systems Department, Plano, Texas.

[17] David Harvey. Ada Field Deployment. *Defense Science and Electronics, Vol 6, No. 2,* February 1987, pp. 27-29.

[18] Rich Hilliard. Personal communication. April 1987.

[19] M.O. Hogan. Personal communication. May 1987.

[20] *IEEE Std 990-1987: Recommended Practice for Ada as a Program Design Language.* January 1987. Technical Committee on Software Engineering of the IEEE Computer Society.

[21] A.J. Jordano. *Managing New Software Technology.* November 1986. Presented at Ada Expo, Charleston, W.Va.

[22] John F. Judge. Ada Progress Satisfies DoD. *Defense Electronics,* June 1985, pp. 77-87.

[23] Lt. Col. John Leary, U.S. Air Force. Personal communication. January 1987.

[24] Dave Miller. *Case History of a Retargetted Ada Compiler.* March 1987. MITRE Corporation, Bedford, Mass.

[25] Donn Milton. The Future of Ada Technology. *Defense Electronics,* December 1986, pp. 47-50.

[26] Charles Z. Mitchell. Engineering VAX Ada for a Multi-Language Programming Environment. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Development Environments,* January 1987, pp. 49-58.

[27] Freeman Moore. Personal communication. April 1987.

[28] J.C.D. Nissen and B.A. Wichmann. Ada-Europe Guidelines for Ada Compiler Specification and Selection. *Ada Letters,* March/April 1984, pp. 50-62. Originally published in October 1982 as a National Physical Laboratory (United Kingdom) Report.

[29] J.C.D. Nissen and P.J.L. Wallis. *Portability and Style in Ada.* Cambridge University Press, Cambridge, UK, 1984.

[30] Patricia Oberndorf. Personal communication. January 1987.

[31] Donald J. Reifer. *Ada's Impact: A Quantitative Assessment.* Technical Report # RCI-TN-255, Reifer Consultants, Inc. Torrance, Calif., March 10, 1987.

[32] Dennis Rowe. *Ada Usage Survey.* January 1987. MITRE Corporation, Washington D.C.

[33] Raymond Szymanski. Personal communication. March 1987.

[34] Valentin Tirman. Personal communication. April 1987.

[35] Will Tracz. Software Reuse: Motivators and Inhibitors. *Proceedings of IEEE COMPCON, San Francisco.* February 1987.

[36] Nelson Weiderman, Mark W. Borger, Andrea L. Cappellini, Susan A. Dart, Mark H. Klein, Stefan F. Landherr. *Ada for Embedded Systems: Issues and Questions.* Technical Report SEI-87-MR-2, Software Engineering Institute, 1987.

[37] Nelson H. Weiderman, A. Nico Habermann, Mark W. Borger, Mark H. Klein. A Methodology for Evaluating Environments. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Development Environments,* January 1987, pp. 5-14.

Index

AD-A182023

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-87-TR-9	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD/TR-87-110
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST.	6b. OFFICE SYMBOL (If applicable) SEI
7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE	
6c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE HANSCOM, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE	8b. OFFICE SYMBOL (If applicable) ESD/XRS1
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-85-0003	
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
63752F	N/A	N/A	N/A

11. TITLE (Include Security Classification) Ada Adoption Handbook: A Program Mgr's Guide	
12. PERSONAL AUTHOR(S) John Foreman John Goodenough	
13a. TYPE OF REPORT FINAL	13b. TIME COVERED FROM _____ TO _____
14. DATE OF REPORT (Yr., Mo., Day) May 1987	15. PAGE COUNT 77
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Ada Adoption Handbook provides program managers with information about how best to tap Ada's strengths and manage this new software technology. Although the issues are complex, they are not all unique to Ada. Indeed, many of these issues must be addressed when using any language for building sophisticated systems. The handbook addresses the advantages and risks inherent in adopting Ada. Significant emphasis has been placed on providing information and suggesting methods that will help program and project managers succeed in adopting Ada across a broad range of application domains.

The handbook focuses on the following topics: program management issues including costs and technical and program control; compiler validation and quality issues; the state of Ada technology as it relates to system engineering; the application of special purpose languages; issues related to mixing Ada with other languages; possible productivity benefits resulting from software reuse; and implications for education and training.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL H. SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630
	22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE

END

8-87

DTIC